
The Design and Development
of a
Simple Diagnostic Expert System

By David Andreae

Department of Computer Science
University of Canterbury

1986

COSC 460

Computer Science Honours Project

University of Canterbury

The Design and Development
of a
Simple Diagnostic Expert System

By David Andreae

1986

Supervisor : Dr. Wolfgang Kreutzer

--- Contents ---

1. Introduction	1
2. Mailmerge	2
3. The Nature of the Problem	6
3.1 The Structure of a Diagnosis	6
3.2 Experiments with a "Test-Client"	7
3.3 The Expert's Knowledge	10
4. Requirements of the System	12
5. Other Expert Systems	14
6. Knowledge Structure and Control	17
7. MERMEX	22
7.1 Overview	22
7.2 Facts	23
7.3 "Facts-to-Facts" Rules	24
7.4 "Facts-to-Actions" Rules	26
7.5 Asking Questions	27
7.6 Different Types of User	30
7.7 Explanation Facilities	31
7.8 Detecting Inconsistent Answers	33
7.9 "I Don't Know" Answers	35
7.10 Recognising an Unsolvable Problem	37
7.11 An Example Diagnosis	39
8. Evaluation of MERMEX	41
9. Conclusion	45

References

- Appendix-A - Why I did not use YAPS
 - Appendix-B - The MERMEX Reference Manual
 - Appendix-C - The Rules
 - Appendix-D - The Program
-

1. Introduction

Expert Systems are a rapidly growing practical application of Artificial Intelligence techniques, and opportunities for research and development in this field are likely to increase dramatically over the next few years. The purpose of this project was to gain experience in Expert System design by developing a small, but practical, diagnostic system.

For the last three years I have been working at a small firm, Systems Software and Instrumentation Ltd. S.S.I. have a number of clients who use standard software packages such as 'Mailmerge'. Since there are only one or two people in the firm who know how to operate these packages, they are regularly required to spend time diagnosing clients' problems. This is usually carried out over the phone. An Expert System could enable other less experienced employees to answer these calls. Alternatively, the system could be provided with the software package itself, thus enabling clients to solve some of their problems themselves. This report describes how a prototype Expert System shell has been developed to perform diagnoses for the 'Mailmerge' program.

2. Mailmerge

The Mailmerge software package [16] was chosen as a suitable domain because it is relatively simple, and yet is a common cause of problems for clients. Mailmerge is a document-formatter used in conjunction with the 'Wordstar' word-processor. One of its primary uses is to enable several copies of a letter to be printed with variable data, such as names and addresses. This data is stored in a separate file, and read by Mailmerge at print-time. Letters such as these are normally called "form-letters". Most of the features in Mailmerge are invoked by inserting 'dot-commands' into the document-file. Although there are only a small number of dot-commands provided, the scope for misuse is considerable. This is due to the many different types of document, and the large number of combinations in which the commands can be used. Figure 2.1 below gives a brief description of some of the more important dot commands available.

.RV var1, var2, ...	Causes data from a data-file (such as names & addresses) to be read into the specified variables. A document containing a .RV command will be repeatedly printed until the data is used up.
&varname&	Inserts the value of a variable into the text.
.DF filename	Specifies the name of the data-file.
.FI filename	Causes another file to be inserted into the document.
.SV var text	(Set Variable) Assigns the text-string to a variable.
.AV	(Ask Variable) Asks the operator to enter a value of a variable.
.PL n	Sets the page-length.
.MT n	Sets the top-margin size.
.MB n	Sets the bottom-margin size.
.PA	Starts a new page.
.OP	Causes page numbers to be omitted.
.PN n	Sets the page number.

Since the Mailmerge commands form a kind of very small 'programming language', and because clients are typically non-programmers, they often find it very difficult to use the dot-commands correctly. A few examples of the kinds of mistakes which can be made are given below. I have not given much explanation as to what these mean, but hopefully they will illustrate the general nature of the different problems that can occur.

Error : Variable name (in '&.&') spelt incorrectly, or has no value.
Observations : '&.&' printed literally in the document. (sometimes described as "rubbish printed" by a client).

Error : Putting .AV (eg. for entering date) in the letter-file, not the command file.
Observations : System asks for date to be entered for every letter.

Error : Too many commas (representing null fields) in a data line.
Observations : Variables are missed out, get incorrect values, and so on.

Error : Missing carriage-return at the end of an inserted-file is at the end of a form-letter.
Observations : Missing pagebreaks between form-letters, and a '.PA' printed at the end of the last line of the letter.

Error : Trailing blanks at the end of the form-letter.
Observations : The dot-command from the top line of the form-letter-file are printed at the top of all letters except the first. (plus any other effects due to these dot-commands not being performed).

2.2 A Sample Diagnosis

Originally I had intended to include a wide selection of Mailmerge knowledge into the system, but in a fairly limited amount of detail. However, it soon became obvious that producing a small subset of the knowledge, in much more detail, was a better approach. This meant that I could become my own expert in that one sub-domain. In fact, the final prototype system only contains knowledge about a subset of problems involving pagebreaks.

The sample diagnosis below illustrates the types of questions that are asked for a simple pagebreak problem.

```
-----  
EXPERT   : Hello! What is your problem?  
  
SYSTEM   :   My Mailmerge document isn't printing  
            pagebreaks correctly.  
  
            What kind of document are you printing?  
  
            I'm printing form-letters.  
  
            Are there extra unwanted pagebreaks, or are  
            pagebreaks being missed out?  
  
            Pagebreaks are missing.  
  
            Are you printing multipage form-letters?  
  
            Yes  
  
            Are the pagebreaks missing between each letter?  
  
            Yes  
  
            Is a "PA" being printed after the last line of  
            the letter?  
  
            Umm ... Yes
```

Go into Wordstar and look at the last line of
the letter-file. Is there a "PA" there?

Yes

Is there a "." in front of the "PA"?

No

SOLUTION : If you put a dot in front of the "PA" then
the pagebreaks will be printed correctly.

3. The Nature of the Problem

3.1 The Structure of a Diagnosis

The development of an Expert System is only possible after carrying out a reasonably thorough analysis of the knowledge and expertise required by the human expert. Such an analysis is often very difficult, since in many cases the human expert is unable to clearly identify how he performs his task. Therefore, gaining insight into the nature of the expertise is not usually possible simply by asking the expert questions. A more successful technique is to record and analyse details of some sample diagnoses.

My first approach involved pretending to be a client with a problem, and asking the expert to find the solution by asking me questions. The example diagnosis in Section-2 was based on one of these tests. Obviously this method could not be particularly realistic, but it did give an indication of the overall structure of a diagnosis, and the types of questions that are asked, as is discussed in the following paragraphs.

Information acquired by the expert during a diagnosis is usually in one of three categories : observations of the printed output, information about the document-file, and 'background' information. Observations of the output can be thought of as 'symptoms', as in a medical diagnosis, and these are often described by the client in a rather vague or misleading manner. During the early part of a diagnosis the observations tend to be fairly general descriptions of the problem. It is the expert's task to narrow down these observations so that more precise details can be obtained. For example, the client might say "I'm getting rubbish printed at the end of my file". After a sequence of questions the expert might obtain a more detailed description such as "The pagebreak at the end of each form-letter is being missed out, and a 'PA' is being printed on the last line, with blanks in front of it". The most substantial part of the diagnosis is simply identifying what the problem is, rather than actually finding the explanation.

Once the description of the problem has been determined in sufficient detail, the expert can usually form some hypotheses of what the error could be. He then needs to ask questions about the document file to confirm or deny these hypotheses. If a suspected error is observed in the document-file, then the problem is solved. For example, the last few questions in a diagnosis might be something like the following :-

...

"Is the last part of your form-letter inserted from another file?"

YES

"Load that file into Wordstar, and move the cursor to the end of it using a ^QC.

Is the cursor one line below the last line of text?"

YES

"Are there blank spaces on the left of the cursor?"

YES

SOLUTION : "You have trailing blanks at the end of the file. Do a ^Y to
delete these, and this should fix your problem."

The overall structure of a Mailmerge diagnosis, therefore, seems to consist of asking questions which will narrow down observations of the output and document-file until an actual error is observed. This is slightly different from many other Expert System domains, in which a solution is not necessarily confirmed at the end of the diagnosis, but is just an 'opinion' based on a whole selection of symptoms. The issue of 'justification' [3] is therefore much more important in these systems than in the Mailmerge problem.

The expert may also need a certain amount of 'background' information. This might include the client's level of experience, the kind of document being printed, and so on. This information is usually obtained at the beginning of a diagnosis.

3.2 Experiments with a 'Test-Client'

One of the more successful experiments performed was to create a typical Mailmerge document, insert a mistake in it, and then ask one of the S.S.I. employees (a non-programmer with very limited Mailmerge experience) to act as a 'client', with the expert asking him questions. After a few of these diagnoses I introduced a restriction that the test-client could only answer "yes", "no", or "I don't know", since it is unlikely that my initial Expert System would have an English language interface. The expert found this much more difficult due to the lack of feedback from the client. English responses usually give more information than the expert actually asks for, and this can act as a triggering mechanism for asking further questions. With yes/no responses there is no

opportunity for following up some piece of information volunteered by the client. Since every piece of information must be explicitly requested (and remembered) by the expert, the yes/no restriction considerably increases the mental effort required. The expert said afterwards that "it required the same knowledge, but a different algorithm".

It was interesting to note, however, that the English diagnoses were no more successful in finding a solution. In fact, the responses seemed to result in more confusion, due to misleading answers. The yes/no diagnoses required more effort, but actually solved the problem in a more organised and precise manner. They also took less time and fewer questions than the English diagnoses, since the expert kept to a fixed line of reasoning, rather than being side-tracked by some irrelevant observation or comment. Another reason was that questions needed to be worked out much more carefully so that a yes or no answer would be possible. Similarly, the client needed to think about his answer more carefully.

One of the most important results of these experiments was to illustrate how unreliable clients' answers can be. Sometimes this was due to misunderstanding the question, perhaps because of unfamiliar terminology. However, instead of asking the expert to explain the question, the test-client often just made a guess at what the expert meant. Clients often seem to avoid giving "I don't know" answers, and this can lead to much confusion for the expert, due to the resulting inconsistencies.

Another problem is that clients often give answers based on what they think *should* be observed, rather than the true situation. For example, when the test-client was asked "Is there a '.PA' at the end of the file", he answered "yes", when really there wasn't. In fact, the missing .PA was the solution to the problem. This is why the expert may need to ask the client to read out, character by character, particular lines of the document-file or the printed output, rather than asking more general questions. This is especially true for questions such as "Do the quotes match?", or "Do the two lines of data have the same format?". Since the clients are not usually programmers, they seem to find this type of question surprisingly difficult to answer correctly. Usually they will just answer "yes", based on the assumption that the data is correct. Consequently this form of question is sometimes worse than useless.

The wording of questions is also very important. For example, in one real-life diagnosis the expert asked "Is the cursor underneath the last line of the file?". The client interpreted this as meaning "on the same line" since the cursor (an underscore character) did seem to be "underneath" the characters it was marking.

An example of how clients do not report observations that they think are unimportant occurred when I inserted an extra comma into a line of address-data, hence creating a null field (eg. "a,b,c,d" --> "a,b,,c,d"). When asked "are there four items in the data line?" the test-client answered (in English) "yes". Afterwards he said that he had in fact noticed the two consecutive commas, but did not think it was worth mentioning. As a consequence of this problem, it is often necessary to ask redundant questions to confirm previous answers or to help resolve some inconsistency. For example, the two questions shown below were both asked by the expert even though they are both asking the same thing :-

- (1) "Is there a carriage-return at the end of the file?"
- (2) "Do a ^QC to get to the end of the file. Is the cursor one line below the last line of text ?"

If a "NO" had been given for the first question, then a solution could have been found without needing to ask the more complicated second question. A "NO" would have been treated as a reliable answer, but a "YES" needed to be confirmed by asking a second, more precise question.

From these experiments it was clear that clients' answers cannot be treated as factual information, but only to provide a rough guide for suitable questions to ask next. Since the final solution is simply a particular observation of the document file, once this found the unreliability of earlier answers does not really matter.

Occasionally the test-client answered "I don't know" to a question. In this situation the expert either reworded the question, or split it into several simpler, more specific, sub-questions. An example of the former occurred when the client was asked "Are there missing pagebreaks?". The expert reworded the question to say "Does the second letter start on the same page as the first letter?".

Sometimes the client realised, after several further questions, that he had given an incorrect answer earlier in the diagnosis. With the 'yes/no' restriction there was no way he could let the expert know of his mistake. Therefore, a Mailmerge Expert System should provide some way of reviewing, and correcting, previous answers.

Another result of this experiment was that it illustrated that there really was a need for an Expert System. The human 'expert' is really an expert programmer with experience in using Mailmerge, rather than an expert at diagnosing Mailmerge problems. In fact, most of the test problems were

not actually solved, although he could probably have solved all of the problems very easily had he been able to see the document himself. Identifying the error just by asking questions is much more difficult. Although clients do not phone up with problems sufficiently regularly for the expert to become a 'real' expert, they occur frequently enough to indicate that clients must have quite a large number of problems which they do not call S.S.I. about, but just try to solve themselves. Therefore, an Expert System provided with the Mailmerge package could save considerable time for both the company and the clients.

3.3 The Expert's Knowledge

The experiments described above made it possible to identify specific aspects of the knowledge required by a human expert. These are discussed in the following pages.

Experience with using Mailmerge is obviously essential. The expert needs to know what the Mailmerge commands do, and how to use them for different types of document. Since a diagnosis consists largely of forming hypotheses about what kind of mistake in the document could have caused the incorrect output, the expert also needs to have considerable knowledge about the relationships between document-errors and observations. This may include simply remembering typical error-to-observation relationships, but will also require a deeper understanding of how Mailmerge works, so that new, unfamiliar problems can be solved. The frequencies of problems and errors also forms an important part of an expert's knowledge since this provides a way of comparing hypotheses. An error which occurs regularly is worth investigating before other less common errors.

Clients are not usually programmers, and hence they often interpret their problems differently from the expert. An observation which could be very important to the diagnosis may be ignored, or not even noticed, by the client (as discussed earlier). Clients may have formed their own idea of what the solution might be, and will therefore tend to interpret observations to fit their own hypotheses.

This can prove very misleading for the expert, and therefore some degree of knowledge about how clients (mis)interpret things is quite valuable. Obviously this will vary considerably from client to client, and so the expert may need to be a good judge of character to perform a successful diagnosis. Clients' differing levels of Mailmerge experience also affect which hypotheses are most likely. A more experienced client is unlikely to have missed out a '.PA' at the end of a form-letter, while this error may be quite common amongst novice Mailmerge users.

The expert should be able to detect inconsistencies in the information obtained from the client.

Incorrect answers may be due to lack of Mailmerge experience, misunderstanding the question, or even (not uncommonly) because they feel that the correct answer would make them look foolish. Knowledge of how to resolve inconsistencies, such as by asking 'trick' questions, is therefore quite important.

Since it is desirable that as few questions as possible need to be asked, knowledge about how much information a question is likely to give is also important. This involves choosing a question which will either give the most positive support for a small number of hypotheses, or the most negative support for a large number of other hypotheses. Additionally, questions should be worded in such a way that the (useful) feedback from the client is maximised.

A general, overall strategy for performing a diagnosis is another important part of an expert's expertise. This involves (perhaps sub-consciously) knowledge about how the above knowledge is used. This can be called 'meta-knowledge' [3].

All of the above enables the expert to ask questions which will be most likely to lead to a final solution. A summary of this is given in figure 3.3.1.

Information acquired during a diagnosis :-

- Observations of the printed output.
- Observations of the document file.
- Miscellaneous 'background' information.

Knowledge required by the expert :-

- Mailmerge commands and their uses.
 - Relationships between errors and expected observations.
 - The way Mailmerge works.
 - Frequencies of problems and their causes.
 - How observations can be misinterpreted.
 - How to ask questions which will obtain the most information.
 - Judgement of clients' experience and style.
 - How to detect inconsistent answers.
 - How to resolve inconsistencies.
 - A 'strategy' for evaluating hypotheses
-

Figure 3.3.1

4. Requirements of the System

This section summarises the main requirements and assumptions (as discussed in the previous sections) on which the development of a Mailmerge Expert System should be based.

- (1) It should be possible to use the system while questioning the client over the phone, or operated by the client directly. Interaction should be fast and simple, hence English-type answers are probably not desirable, except maybe for the initial problem description, or for volunteering information. Yes/No or Menu type questions are easier to implement, and normally more convenient for the user.
- (2) The knowledge structure should be such that rules (or whatever) can be added easily by someone other than the knowledge engineer (ie.myself). If this is not possible then the system will rely on an expert knowledge engineer to be available, thus losing much of the advantage of Expert Systems. This requirement also means that a relatively 'straightforward' technique for acquiring knowledge from the Mailmerge expert should be developed.
- (3) The system should probably be designed as a shell-type of program, rather than specifically for Mailmerge. Mailmerge was only chosen as a simple example of a typical software package. Ideally the system should be able to be extended to other domains which have a similar nature.
- (4) "I don't know" answers should be catered for.
- (5) Inconsistent answers should be able to be detected and resolved.
- (6) Sometimes a client realises later that an earlier question was answered incorrectly. Therefore there should be provision for a user to look at, and correct, any previous answers.
- (7) There should be some sort of explanation facility so that the user can see the current most likely hypothesis as a way of explaining why a particular question was asked. This might also enable a user to actually solve the problem on her own, rather than continue the diagnosis.

- (8) Clients with different levels of experience may need to be asked different types of questions. Some sort of facility for this should be provided.
 - (9) The dialogue should be clear and coherent. This is especially important if the system is being operated over the phone, since the operator will be essentially reading out the questions appearing on the screen.
 - (10) The system should recognise when it cannot solve a problem, and display an appropriate message.
-

5. Other Expert Systems

There are currently a very large number of Expert Systems which have been built for a wide variety of applications. This section gives a very brief description of a few of the systems which have relevance to the Mailmerge problem.

MYCIN (Shortliffe 1976)

MYCIN is one of the earliest, and most well-known Expert Systems for providing consultative advice on diagnosis and therapy for infectious diseases. Knowledge is encoded as production-rules, each of which contains premise clauses typically in the form "(predicate, object, attribute, value)". The strength of association between the premise and action clauses in each rule is specified with the use of 'confidence-factors'. Backward-chaining is used for the basic control strategy, resulting in an exhaustive depth-first search of the AND/OR goal tree. MYCIN is organised so that each subject is effectively exhausted as it is encountered, and this tends to group together questions about a given topic. In other words, when a subject is encountered (such as identifying a particular infectious organism) a generalised goal is set up so that all of its attributes are collected before continuing with other goals. This results in a much more focussed and methodical system, but may collect information which is not strictly necessary. An interesting conclusion made by Cendrowska (1984) is that much of the effectiveness of MYCIN is due to fine-tuning and ad hoc modifications, rather than the use of backward chaining and production rules. (references [9], [1], [2]).

INTERNIST (Pople & Myers 1977)

Internist is a diagnosis program in the domain of internal medicine. One of its main goals was to model the way clinicians do diagnostic reasoning. It explores ...

- the way that certain symptoms evoke particular disease-hypotheses in the mind of a clinician.
- how hypothesised diseases give rise to expectations of other symptoms.
- how clinicians focus on particular disease areas and temporarily ignore other symptoms that they judge to be irrelevant.
- how clinicians decide between competing disease hypotheses.

Internist uses a 'disease-tree' to represent its knowledge, with leaf-nodes representing actual diseases, and the higher-level nodes representing more general disease areas. Each of the actual diseases has an associated list of expected symptoms, and these are percolated up the tree (prior to

diagnosis) so that each general disease area is associated with the intersection of the symptoms of that node's offspring. This information is used during consultation for selecting a disease area on which to focus. The disease-tree provides a means of narrowing down the hypotheses until one is found which accounts for all of the symptoms. Internist is purely associational. It does not attempt to model any disease process, but considers a disease to be a static category, and diagnosis as the task of assigning a patient to one or more of these categories. (references [1], [2], [3])

MUD (Kahn & McDermott 1985)

The MUD system is a fairly recently developed drilling-fluid diagnostic and treatment consultant for 'mud engineers'. The designers of MUD used an 'evidential' approach, rather than a 'causal' approach to diagnosis. This means that instead of reasoning in terms of a causal model, or an explicit representation of how hypothesised causes bring about symptoms, MUD explicitly represents the weighting of evidence to diagnostic conclusions. The intermediate steps in the causal path from a hypothesised solution to the expected evidence is not represented. This evidential approach has the following structure :-

- (1) Generate a set of plausible hypotheses.
- (2) Order the hypotheses for investigation.
- (3) For each hypothesis determine the relevant evidential considerations.
- (4) Accept or reject each hypothesis.

MUD follows this structure fairly closely, as do many other diagnostic systems (including Internist, described above). One aspect of the MUD system which is worth noting is the assumption that all data entered is certain. Apparently this has not degraded MUD's performance, firstly because there are typically several significant observations which can evoke a hypothesis, and secondly because small errors in some fraction of several evidential considerations may not have much effect on the final acceptance or rejection of a hypothesis. (reference [4])

PIP (Pauker et al. 1976)

The Present Illness Program, or 'PIP', is a medical diagnosis system based on a network of frames. The frames contain data such as typical findings, relationships with other frames, and rules for judging how well a set of findings exhibited by a patient matches the situation described by the frame. This is stored in slots such as "IS-SUFFICIENT", "MUST-HAVE", "CAUSED-BY", "COMPLICATION-OF", and so on. The key strategy in the diagnosis is the matching of findings with those indicated in a disease frame, and the selection of frames which cover all of the findings. The reasoning process first acquires a new finding by asking some

question. All the facts relevant to this finding are then located, and the list of active hypotheses is updated. This is achieved by checking slots such as "MUST-NOT-HAVE" and "IS-SUFFICIENT", and by scoring the hypotheses with the scoring rules included in each frame. The highest rated hypothesis then becomes the focus of attention, and a question is generated for the next unexplored finding. One problem with PIP has been that questioning can be somewhat erratic, due to the top-ranking hypotheses alternating rapidly. (reference [1])

EXPERT (Weiss & Kulikowski 1979)

EXPERT is not an Expert System itself, but a general-purpose language and notation for representing expert knowledge. The representation is consistent with the typical diagnostic structure of interpreting a set of observations, and selecting and investigating an appropriate hypothesis. One application of EXPERT has been a consultation system for rheumatic diseases. An EXPERT model consists of three sections : hypotheses, findings, and decision-rules. There are three types of decision rules for describing the logical relationships among findings and hypotheses :-

- FF - Finding-to-Finding rules
- FH - Findings-to-Hypotheses rules
- HH - Hypotheses-to-Hypotheses rules

FF-rules specify the truth-value of findings that can be deduced directly from already established findings. FH-rules are a logical combination of findings which indicate confidence in the confirmation or denial of hypotheses. HH-rules enable the model builder to specify inferences among hypotheses. Instead of using a backward-chaining mechanism for production-rule evaluation, as in MYCIN, EXPERT evaluates rules in an ordered fashion that has been prespecified by the model designer. It is also possible to form groups of rules to be part of a 'questionnaire', thus avoiding the need for a complicated control mechanism.

(references: [4], [1], [3])

6. Knowledge Structure and Control

The most important part of developing an Expert System is deciding on the underlying knowledge representation, and control mechanism. This requires a clear understanding of the nature of the domain knowledge, and a reasonably detailed description of what the system should be able to do. Unfortunately, defining a suitable knowledge representation can be one of the most difficult parts of the development process since the nature of the domain knowledge may only become apparent *during* the development process, rather than before. Similarly, the requirements of the system cannot usually be specified beforehand, since they constantly change during development. This is one of the differences between building an Expert System, and writing most conventional programs [1].

The next few pages give a very brief overview of how a suitable knowledge representation and control mechanism were eventually chosen for the Mailmerge problem.

Originally it seemed that the most important knowledge required in the system would be the relationships between errors in the document, and the corresponding observations of the printed output. These 'error-to-observation' (or 'disease-to-symptom') relationships are characteristic of most diagnostic symptoms, as was discussed in Section-5. My first design attempts based on this used a frame-like system, similar to PIP, with slots such as "CAUSE-OF", "MUST-OBSERVE", and so on. A more rule-oriented system, similar to the MUD system, was also investigated, but this had essentially the same underlying organisation. Figure 6.1 illustrates this 'error-to-observation' structure. When an observation is found, this will trigger any relevant error-hypotheses, which are then evaluated according to the evidence supporting it. The highest scoring error is investigated further by asking questions about other expected observations.

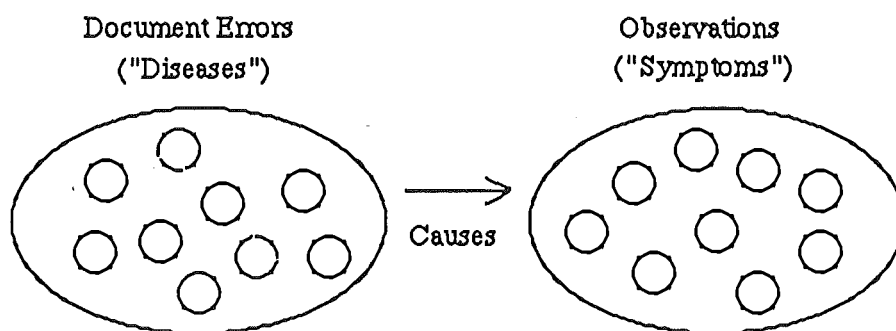


Figure 6.1

The most important problem with this method is the difficulty of finding a suitable scoring strategy for choosing the best hypothesis, and the next question to ask. In a human expert this evaluation process may be very complex, and simulating this using some general numerical scoring strategy is probably impossible, although there are many Expert Systems, such as MYCIN, which use this method and perform reasonably successfully. However, they usually employ additional control strategies or ad hoc techniques, to make the system effective [9].

With a numerical scoring strategy it is usually very difficult to control or predict how the system will behave, since the declarative error-to-observation relationships (in which the scoring is specified) do not explicitly reflect the structure of a diagnosis. In the Mailmerge system, where an organised, coherent dialogue is quite important, it is desirable for the knowledge to have some clear structure to it, rather than a hidden structure based on obscure numerical weightings.

The next approach was to make the hierarchical (general-to-specific) nature of the Mailmerge knowledge explicit in the knowledge base. This was based loosely on the 'disease-tree' concept in Internist, except that two trees were to be used : an error (or 'disease') tree, and an observation (or 'symptom') tree. The error-to-observation relationships (and numerical scoring) would still be represented as before, but questions to ask the user would be chosen in a more structured manner, as controlled by the tree-structure. Questioning would progressively narrow down the relevant error and observation branches until a leaf-node is found which best explains the given observations. The diagnosis would proceed in a hierarchical manner, starting with a general problem description and error-hypotheses, and continue until the set of observations were detailed enough for the final solution to be found. Figure 6.2 illustrates the overall structure of this approach.

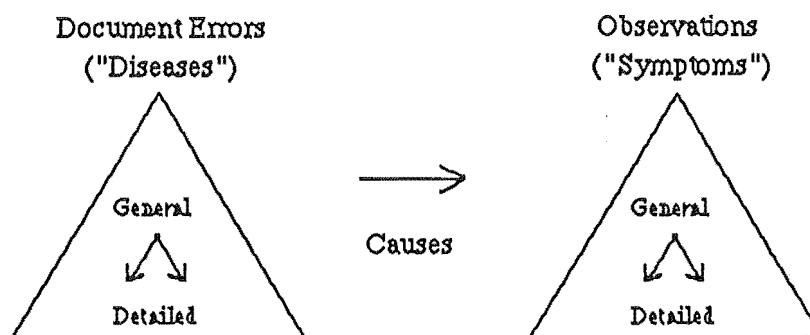


Figure 6.2

Unfortunately, developing the details of this (rather vague) design-description proved virtually impossible. Finding a meaningful scoring strategy for evaluating the best branches to explore was no less difficult than the previous method. For example, assigning numerical weights to a general error such as "missing .PA" is almost pointless since this error can cause so many different effects, depending on the particular situation and the interpretation of the problem by the user. Relationships between errors and observations are not precise enough for numerical weightings to be used effectively. It is certainly the case that different observations have different likelihoods of occurring in particular situations, but it was found that these priorities could only be represented clearly using explicit ordering.

A more important problem with the tree method is that the knowledge cannot really be represented in a tree form without considerable redundancy, as was found when attempting to develop a tree structure (using lots of little pieces of paper). This is because most observations and errors can have several parents, due to many possible ways of interpreting and classifying the knowledge.

It soon became apparent that the whole approach of basing the system on the fairly low-level, purely declarative, error-to-observation relationships was inappropriate for the Mailmerge system.

The method which has proved much more successful is to drive the diagnosis in a forward-chaining manner, from observations themselves, rather than working back from explicit knowledge about errors using an indirect and confusing scoring strategy. In other words, the knowledge in the system consists of rules in the form :-

"If a, b, and c have been observed, then
investigate whether x and y have been observed

If several rules have the same, or similar, left-hand side conditions then the *order* of the rules can be used to indicate priority. This is preferable to using some sort of numerical scheme, since there are no problems with inconsistent interpretation of the values (which is particularly relevant in a hierarchical knowledge structure), and can be much clearer and simpler for the knowledge engineer and the expert.

One of the important differences between the Mailmerge system and many other expert systems is that the final solution is not an 'opinion' based on a set of observations, but is the actual observation of an error in the document file. In other words, any hypotheses can be validated by examining the document-file. At the start of a diagnosis only a general problem description is known, such as "*missing pagebreak*". The user is asked progressively more and more specific

questions until a detailed set of observations is obtained. Questions are asked according to how useful the information will be, rather than for directly supporting or rejecting particular error-hypotheses. During this process, whenever there is sufficient evidence for it to be worthwhile investigating some particular document error, then the appropriate question or questions about the document-file are asked to confirm or deny this error-hypothesis. Figure 6.3 illustrates this process, in general terms on the left, and for a specific Mailmerge example on the right.

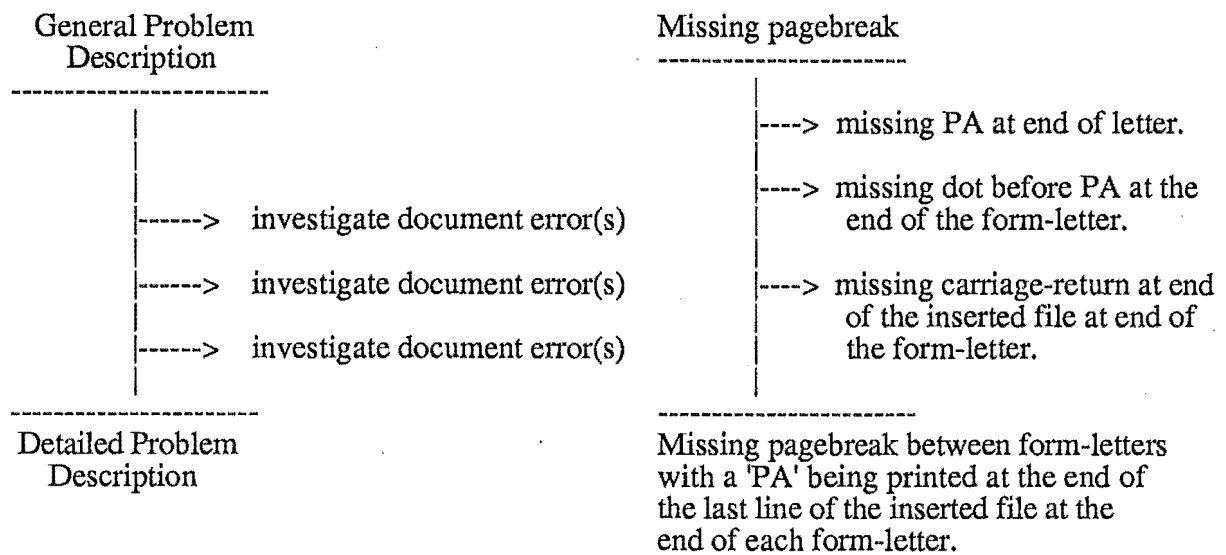


Figure 6.3 - The Structure of a Diagnosis

Rules which directly reflect this structure make controlling and predicting the behaviour of the system much simpler, which is an advantage for ensuring a coherent dialogue. This is because more of the control is built into the rules themselves, since they define what to *do* in a given situation, as opposed to the "error-to-observation" rules which are purely declarative. Each rule has *implicit* knowledge about the error-to-observation relationships, which enables the choice of question in a given situation to be explicitly stored in the knowledge base. In other words, most of the knowledge acquired from the expert consists of 'observation-to-action' (or 'stimulus-response') relationships, as opposed to storing lower-level knowledge with a 'clever' inference mechanism to model the expert's thought processes. However, this does not mean that the system will just be like a conventional program, since the rules are, on the whole, independent of each other. The work on the Mailmerge system so far has suggested that this compromise between declarative and procedural knowledge is more sensible than attempting to make it totally declarative, with a simplistic inference mechanism performing all of the control. A comparison of declarative and

procedural representations can be found in [8].

Another alternative, still in the early research stage, is to include 'meta-knowledge' to represent the reasoning process, rather than putting it in the inference mechanism itself. This is discussed by Hayes-Roth et al. [3].

The overall structure for the above approach is illustrated in figure 6.4. This may not be appropriate for many other diagnostic applications, but it seems to be quite effective for the Mailmerge problem.

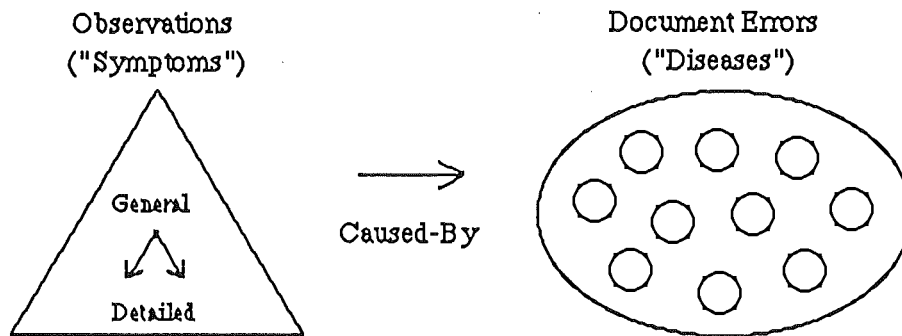


Figure 6.4

The first 'working' system based on the above structure was written in the production-system YAPS [17] (which is quite similar to OPS5 [19]). After writing enough of the program to show that this method could be quite successful, I abandoned YAPS (for reasons discussed in Appendix-A) and wrote my own system, MERMEX, using Lisp. This is described in the following section.

7. MERMEX

7.1 Overview

MERMEX is an Expert System shell designed for applications with a similar structure to the Mailmerge diagnosis problem. The system is based on *facts* and *production-rules*, where the production-rules operate in a forward chaining manner [10] with priority indicated by the order in which they were loaded into the system. Knowledge consists primarily of rules which subdivide an initial general problem description into a detailed set of observations, by asking questions either in a yes/no form, or a menu structure.. The final solution is obtained when an actual error in the document-file is observed. Figures 6.3 and 6.4 in the previous section illustrate this process diagrammatically.

As well as providing the basic knowledge representation and control mechanism, MERMEX also includes a number of other features. One of these is the ability to detect inconsistent answers. These can be reported to the user, who is then given the opportunity of correcting them. There is also a facility to enable the user to see the list of previous answers, which can then be changed or corrected.

MERMEX employs a simple, but effective, technique which will automatically deal with "I don't know" responses, in the situation where explicit rules for coping with these have not been included in the knowledge-base.

A simple explanation facility is also provided which allows the user to answer a question with the response "why". This will cause a message to be printed describing the current hypothesis.

These, and other features, will be explained in detail in the rest of this section. Some of the features are illustrated in the example diagnosis given in Section 7.11. There is also a Reference manual included in Appendix-B.

Efficiency in MERMEX is achieved because only those production rules relevant to the most recently acquired facts are examined. This minimises the amount of searching and rule-evaluation that needs to be done.

7.2 Facts

As in many production systems, the most basic form of knowledge is a *fact*. A MERMEX fact is simply an association between a name and a value, as shown in the formal definition below :-

fact ::- (<factname> <value>)

In most MERMEX applications, a typical fact-name would represent some statement about the world, and the fact-value would indicate a measure of belief. For example, in the Mailmerge system most facts are associated with observations of the printed output or the document-file, and most of the fact-values are one of the following four values, since these have special meaning to MERMEX :-

TRUE	-	The statement represented by the factname is true.
FALSE	-	The statement represented by the factname is false.
"?"	-	The user answered "I don't know" when questioned about the fact directly.
nil	-	No information is known yet.

If the <value> part of a fact is one of the first three values then a slightly more convenient notation may be used, rather than the "(factname, value)" format :-

(x)	is equivalent to	(x TRUE)
(NOT x)	is equivalent to	(x FALSE)
(? x)	is equivalent to	(x ?)

Figure 7.2.1 shows several examples of Mailmerge facts, together with their English descriptions.

(form-letters)	:	"The user is printing form-letters"
(NOT msng-pgbrks-btwn-fls)	:	"There are NOT missing pagebreaks between the form-letters"
(? PA-end-fl)	:	"When the user was asked 'Is there a .PA at the end of the form-letter file?' he answered 'I dont know'."
(msng-crgret-end-if 0.7)	:	"There is a 0.7 probability that there is a missing carriage return at the end of the inserted file". (This sort of fact is not actually used in the Mailmerge system).
(looking-at end-fl)	:	"The user currently has the end of the the form-letter displayed on the Wordstar terminal." Note that the fact-name is 'looking-at', and the value is 'end-fl'.
(NOT looking-at top-fl)	:	"The user is NOT currently looking at the top of the form-letter-file."

Figure 7.2.1 - Some Mailmerge Facts

7.3 "Facts-to-Facts" Rules

Facts-to-Facts rules, or 'FF-rules', provide a means of directly inferring new information from existing information. When the facts on the left hand side of the FF-rule (surrounded by an implicit AND) all become true, then the list of facts on the right hand side are asserted into the knowledge-base. There are no priorities associated with FF-rules. They simply fire as soon as the premise conditions are satisfied. For example ...

```
(FF (it-is-a-bird)
    (NOT it-is-an-ostrich)  -->  (it-flies)
                                (it-makes-tweet-tweet-noises)
                                (NOT it-eats-icecream))
```

The list of facts in the premise of an FF-rule can actually be a list of logical expressions. The right hand side, however, is restricted to being a single list of facts. The following (rather meaningless) example illustrates this :-

```

      (FF (AND (x)
                (OR (NOT y) (z)))
        (OR (? a)
            (b 2) (c (+ b 8))
            (AND (c) (NOT (d)))    ---> (p) (NOT q)
                                           (message "hello")))

```

Note that the expression "(NOT (d))" in the above example does not mean "d is false", but rather "d is not known to be true". The formal syntax of an FF-rule is as follows :-

```

<FF-rule>      ::-      ( FF <fact-exprs> <facts> )

<fact-exprs>   ::-      <fact-expr> <fact-expr> ...

<fact-expr>    ::-      <fact>
                        | ( AND <fact-exprs> )
                        | ( OR  <fact-exprs> )
                        | ( NOT <fact-exprs> )

<facts>        ::-      <fact> <fact> ...

```

As well as inferring new information, FF-rules serve another important function, namely to detect inconsistent answers. However, the discussion on this is left until section 7.8.

FF-rules can also operate in the reverse direction. For example, when the following rule is loaded into the knowledge base ...

```

      (FF (it-is-a-mouse)  -->      (it-squeaks)
                                           (it-eats-cheese))

```

... MERMEX will also generate the rules ...

```

      (FF (NOT it-squeaks)  -->      (NOT it-is-a-mouse))
      (FF (NOT it-eats-cheese) -->      (NOT it-is-a-mouse))

```

In a situation where reverse-rule generation is undesirable, the production rules can be defined

using the 'FF*' symbol. Note that MERMEX will only generate reverse rules if the premise condition consists of a single fact, since to do otherwise would introduce considerable complexity. Fortunately a large proportion of Mailmerge rules are in this form, and so the reverse-rule feature can often be utilised.

7.4 "Facts-to-Actions" Rules

Facts-to-Actions rules, or 'FA-rules', form the most important part of the knowledge base. FA-rules have a similar format to FF-rules, except that the right hand side specifies a sequence of actions to be performed, rather than a list of facts to be asserted. The purpose of these actions is normally to ask the user a question, or sequence of questions.

The following example shows a very simple FA-rule. The English equivalent of the rule is : "If there are missing pagebreaks, and form-letters are being printed, then find out if the missing pagebreaks are occurring between each form-letter (as opposed to half-way through)." FA-rules can be much more complex, but it has turned out that most of the Mailmerge rules are quite similar to the one below.

```
(FA  (msng-pgbrks)
      (form-letters)    -->  (QUESTION msng-pgbrks-btwn-fls)
                           (QUESTION PA-printed-btwn-fls) )
```

The right-hand-side actions can actually be any lisp function, although normally only the functions provided by MERMEX would be used. The actions have an implicit AND surrounding them, and are therefore performed in sequence until one of them returns a nil value. The QUESTION function shown above will (generally speaking) return a non-nil value if the user answers "yes" to a question. The general format of an FA-rule is as follows :-

```
(FA  <fact-expression>
      <fact-expression>
      ...                --->  <lisp-expression>
                                <lisp-expression>
                                ...
```

If there are several FA-rules ready to fire, the one with the highest priority is chosen first. The priority of a rule is given by the order in which it was loaded into the system, where the first rule loaded has the highest priority.

The production system YAPS [17] (which is based on OPS5) is essentially quite similar to MERMEX. However, the choice of which rule to fire next is based on how recently the facts in the premise were added to the knowledge-base, which is supposed to result in the system focussing on one line of reasoning, hence ensuring a coherent dialogue. Although this would be quite easy to implement in MERMEX, there were several reasons why the simpler rule-ordering method was chosen instead. For one thing, the Mailmerge FF-rules tend to infer new information in a rather 'unorganised' way. Many of the facts inferred may be quite irrelevant to the current sequence of questions, and hence could cause the opposite of the desired focussing effect. The YAPS strategy makes controlling, or predicting, the system's behaviour very difficult, as I found when writing my initial system (the forerunner of MERMEX) in YAPS. For example, in the common situation where several rules have identical premise conditions, it is desirable for these to be fired in an explicit order of priority. In MERMEX this is straightforward, but in YAPS it requires all sorts of 'clever' tricks, or extra conditions in each rule.

Using rule-ordering to indicate priority is quite widely used, and has been shown to be reasonably successful. One example is EXPERT, which was discussed in Section 5. Incidentally, the similarity (at least in name) between MERMEX's FF and FA rules, and EXPERT's FF and FH rules, is not a coincidence.

MERMEX also allows FA-rules to be explicitly marked with a numerical priority (0, 1, 2,...). When some rules are ready to fire, they are first put into their priority group, with 0 being the default, and then each group is sorted according to its loading-order. Rules in group N have priority over rules in group N+1. These numerical values can be used to indicate whether a rule is 'weak' or 'strong', and so rather than having to put weak rules at the end of the knowledge-base, they can be grouped with other similar rules. However, I have not yet used this feature, and will probably remove it, since I do not think numerical values are a particularly appropriate way of organising knowledge. The main reason is the difficulty of maintaining a consistent interpretation of the numbers. It is *relative* priority which is important, and this is achieved much more clearly and easily by the grouping and ordering of the rules.

7.5 Asking Questions

MERMEX provides a variety of functions to enable information to be obtained from the user. For example, the YES-NO command shown below will allow a "Yes", "No", or "?" (I don't know)

response to be entered, and will set the specified fact-value accordingly.

```
(YES-NO msng-pgbrks "Are there missing pagebreaks?")
```

Normally, however, questions are defined within a QUESTION-declaration :-

```
(QUESTION msng-pgbrks
  (YES-NO "Are there missing pagebreaks?"))
```

The QUESTION-declaration associates a list of actions (in this case a single YES-NO command) with a fact-name. These can be executed (with an implicit AND around them) by putting a QUESTION-command, such as "(QUESTION msng-pgbrks)", on the right hand side of an FA-rule. Often these actions will ask a whole sequence of questions, or "questionnaire". The function-value returned by a QUESTION-command will be non-nil if the associated fact-value has the value TRUE when the sequence of actions is completed. If the QUESTION-command has already been performed earlier then the actions are not executed a second time. Additionally, if the fact-value is already known beforehand, this will prevent execution. Note that if any of the YES-NO commands do not have a fact-name as the second parameter, as in the example above, then the fact-name of the enclosing QUESTION is used instead.

There are a variety of other functions also provided by MERMEX for use in QUESTIONS. Some of these are shown in the example below.

```
(QUESTION personal-details
  (ENTER-VALUE name "What is your name?")
  (NO-YES single "Hello " name ". Are you married?")
  (MENU sex-menu)
  (IF-YES
    (ENTER-VALUE children "How many children do
                          you have?"))
  (YES-NO employed "Are you currently employed?")
  (IF (AND (employed) (NOT single))
    (YES-NO partner-employed "Is your " partner-name
                              " also employed?"))
  (QVALUE t)    ;-- another way of returning a value
)
```

```
(FF (sex male) --> (partner-name "wife"))
(FF (sex female) --> (partner-name "husband"))
```

Although most of the questions in the Mailmerge system only require a yes/no response, a few questions at the start of the diagnosis use a menu structure, since this is more convenient for determining the initial general problem description. An example of a typical MENU-definition is given below :-

```
(MENU problem
  (MSG "Which of the following problems are you having?")
  (OPTION pgbrks
    (MSG "Pagebreak problem."))
  (OPTION pgsizes
    (MSG "Incorrect page-sizes."))
  (OPTION datafile
    (MSG "Data-file problems.)))
```

When the command "(MENU problem)" is given in an FA-rule, the following will be displayed :-

```
Which of the following problems are you having?
(1) Pagebreak problems.
(2) Incorrect page-sizes.
(3) Data-file problems.
```

>

The user can then enter one or several of the option-numbers, with a "?" preceding a number to indicate a 'maybe' or 'I dont know' answer. The fact-name associated with each option (eg. 'pgbrks') is set to TRUE, FALSE, or "?".

7.6 Different Types of User

The choice of questions, and the way in which they are asked, may need to be different for users with different levels of Mailmerge experience. The priority of certain rules may also be affected, since a problem which is common for a novice might be unlikely for a more experienced user.

Since the way in which the level of experience is used will depend very much on the particular situation, MERMEX cannot cope with this automatically. Instead there must be explicit conditions in the rules themselves. The Mailmerge system does this by defining three factnames :-

NOVICE, INTERMEDIATE, and EXPERT

One of these is set to be true at the start of the diagnosis by asking the user his or her level of experience (see [18]). Rules can then use these facts as required, as illustrated in the examples below :-

```
(FA (...)
  (...)
  (OR (INTERMEDIATE)
      (EXPERT))      -->  (...) (...) ...)
```



```
(QUESTION x
  (IF (OR (INTERMEDIATE) (NOVICE))
    (QUESTION x1)
    (QUESTION x2)
    (NEWFACTS (x)))
  (IF (EXPERT)
    (YES-NO "..."))
  ...)
```

Unfortunately, coping with different types of user requires considerably more work on the part of the knowledge-engineer, since many more rules and questions must be produced. Therefore, the current Mailmerge system only includes one or two rules such as the above, just to illustrate their use. A possible extension to MERMEX could determine the user's level of experience automatically, perhaps by recording how many incorrect or "I don't know" answers have been given during previous diagnoses, or by the types of problems they have had.

7.7 Explanation Facilities

Many production systems have an explanation facility which displays the rule currently being fired. This is inappropriate for the Mailmerge system (other than for development purposes), since it would be meaningless to a typical user. Any explanations need to be in pre-written English text, rather than something automatically generated by the system.

The first method which MERMEX supports is the "WHY" command, which simply displays the current most-likely solution. This is achieved by associating a 'hypothesis-description' with each rule. For example ...

```
(FA (msng-pgbrk-btwn-fls)
    (PA-printed-btwn-fls)
    --> (HYPOTHESIS msng-crgret-end-if-end-fl)
        (Q inserted-file-end-fl)
        (Q msng-crgret-end-if)

(HYPOTHESIS msng-crgret-end-if-end-fl
  "You are probably inserting a file at the end of the form-letter, and have
  missed out the carriage return at the end of it. This would cause the .PA
  to be printed, and the pagebreaks to be missed out")
```

The hypothesis description can be put directly into the rule, but since several rules may be associated with the same hypothesis it is more desirable to define it separately, as above.

When the user answers "WHY" to a question, MERMEX will just display the associated hypothesis-description, as illustrated below :-

Is the last part of the form-letter inserted from another file ?

> WHY

"You are probably inserting a file at the end of the form-letter, and have missed out the carriage return at the end of it. This would cause the .PA to be printed, and the pagebreaks to be missed out."

Is the last part of the form-letter inserted from another file ?

>

This not only explains to the user why the question is being asked, but also may enable the user to solve the problem herself, without needing further advice from the Expert System. Additionally, this feature means that the system could be used as a kind of advice system, rather than just for diagnosis. In fact, MERMEX could be used to create a reasonably effective 'intelligent' on-line manual, where the goal of the system is to identify what the user wants to know and to print the appropriate information.

Unfortunately, it is necessary that almost every rule includes a hypothesis description if the WHY facility is to be effective. Otherwise MERMEX will just use the most recent hypothesis definition, which may not be applicable. However, this is probably a good requirement since it provides documentation for the knowledge base. If a rule does not specify the associated hypothesis then someone looking at the rule is unlikely to know where the right-hand-side questions are leading to.

Another very useful feature is the "SUMMARY" command, which will display, on request or in a rule, a list of the information obtained from the user so far. When the user is answering a question he can type "SUMMARY", and something like this will be displayed ...

The following is a list of what you have told me so far :

- (1) There are missing pagebreaks in the output.
- (2) You are printing form-letters.
- (3) There are missing pagebreaks between each form-letter.
- (4) There is NOT a missing PA at the end of the form-letter file.
- (5) The last part of the letter is inserted from another file.

Which of these are incorrect ? (answer 0 for none)

>

This simple feature is very useful in the Mailmerge system since it helps users to clarify to themselves what their problem really is. Also, as the example indicates, the user is given the opportunity of correcting any previous answers. A SUMMARY command is executed automatically by MERMEX if it gets into a situation where no more rules can fire. Any corrections made could enable the diagnosis to continue, rather than just declaring the problem as unsolvable. Note that these corrections will result in the whole knowledge-base being adjusted to reflect the

changes. The method for achieving this will be discussed later.

If a large number of questions have been asked, it might be desirable to suppress some parts of the summary. Later versions of MERMEX will hopefully include techniques to do this. One possibility is to make use of the hierarchical nature of the Mailmerge knowledge so that only the most specific interpretation of each observation is displayed. In the above example, (1) is just a generalisation of (3), and therefore need not be included. To determine this, the FF-rules could be examined. For example, in the current system there is an FF-rule like the following :-

```
(FF (msng-pgbrks-btwn-fls) --> (msng-pgbrks))
```

To make the hierarchical structure more explicit, any FF-rule which represents hierarchical inferences could be defined with a different symbol, such as "AKO" (A Kind Of). The SUMMARY mechanism could do some sort of search through these AKO-rules to determine which of the list of questions to include in the summary. Another possibility is to suppress the questions which were answered with 'no', such as (4) above. However, this is not particularly desirable since it is often these questions which are most likely to have been answered incorrectly.

The way in which MERMEX implements the SUMMARY feature is very simple. Whenever the user is asked a question, the fact-name associated with that question is added to a list. The "SUMMARY" command just prints out the English descriptions of these facts. Facts-descriptions can be specified using the DESC definition, as in the example below :-

```
(DESC form-letters "You [are/are not] printing form letters.")
```

Note that if a DESC has not been specified, the SUMMARY command will just display the fact name. However, a system should normally include a DESCription for every fact, since it provides documentation for the knowledge base.

7.8 Detecting Inconsistent Answers

One important characteristic of the Mailmerge problem is that clients often give incorrect responses. However, the only way that MERMEX can detect these is by noticing some inconsistency with other information. Therefore, to enable incorrect answers to be found, it is necessary to ask questions which are not consistent with the current evidence. These weaker-priority 'trick'

questions would normally only be asked when the system cannot think of any alternative questions, since this situation indicates that either the knowledge base is incomplete, or the user has made a mistake.

MERMEX uses FF-rules to detect inconsistencies. Whenever an FF-rule fires causing some fact to be set to TRUE (or FALSE), when its previous value was FALSE (or TRUE), then something must be wrong.

Every fact in MERMEX has a flag indicating whether its value was 'inferred' (from an FF-rule) or 'asked' (using a question). If the most recent answer shows an inconsistency with a fact which was inferred earlier then this earlier fact's value is altered so it is consistent with the new information. Any relevant FF-rules are then fired. Until the actual source of the inconsistency (ie. a question) has been identified, earlier facts are corrected automatically, based on the assumption that recent information is more reliable since the user will have probably gained a better understanding of his own problem. Note that replacing previous fact-values, and firing appropriate FF-rules, does not actually maintain a consistent knowledge-base. This would require an "undo-FF-rule" function to be produced. However, when the source-question causing the inconsistency is found then the knowledge-base can be corrected properly. This occurs when the earlier (inconsistent) fact is marked as 'asked'. The question which obtained this fact, and the most recent question, can then be displayed to the user, using the DESCriptions of the two facts. For example ..

The answer you have just given me seems to be inconsistent with what you told me earlier.

(1) There is not a 'PA' being printed between form-letters.

(2) A 'PA' is being printed at the end of the last line of the form-letter.

Which is correct? (answer '1', '2', 'N'one, or 'B'oth)

>

If a correction is made then all inferred facts are removed from the knowledge-base (simply by incrementing a global 'fact-version-number'), and FF-rules are refired using the new list of corrected answers.

If, on the other hand, the user says that both of the answers were correct, then the knowledge-base must remain inconsistent. To prevent the inconsistency being reported again after later questions, MERMEX will put the fact-name associated with the earlier question into a

'CONFLICT-list'. Later inconsistencies with facts in this list will not be reported after each question. When (and if) MERMEX gets into a situation where no rules can fire, it will display the facts in the CONFLICT-list in a similar way to the SUMMARY command, and again ask the user to check them.

It is somewhat doubtful whether the reporting of inconsistencies to the user is actually desirable. In the example above it is fairly obvious that the first answer was the incorrect one, since it is easy not to notice a 'PA' at the end of a line of text if only a general question such as "Is a PA being printed between the letters?" is asked. It is for this reason that the second question is asked. The second question is itself checking for an incorrect answer, and it is therefore unnecessary to report the inconsistency to the user. A better solution might be to always assume that the more recent answers are correct, and fix up the knowledge-base automatically. If any of these automatic changes were not justified, then the user is still given the opportunity of changing them back again when the SUMMARY command is executed at the end of an unsuccessful diagnosis. Corrected answers could be marked with some appropriate message when the summary is displayed.

The assumption that an answer just received is more likely to be true than earlier answers is not entirely valid, since the user may just make a mistake, or misinterpret that particular question. A solution to this could be to resolve inconsistencies according to how many earlier answers it conflicts with. If a response is inconsistent with several earlier questions then it could be assumed incorrect, and immediately reported to the user. If it only conflicts with one, more general, observation then the earlier answer could be assumed incorrect, and changed automatically. Whatever solution is chosen it will not be perfect for every situation, and may need to be combined with specific 'inconsistency-information' associated with those rules which require a different technique.

7.9 "Don't Know" Answers

Sometimes a client may answer "I don't know" to a question. One situation when this could arise is if the required information is unavailable. However, this is assumed not to occur in the Mailmerge problem, since the output text and document file, to which most questions refer, must always be available.

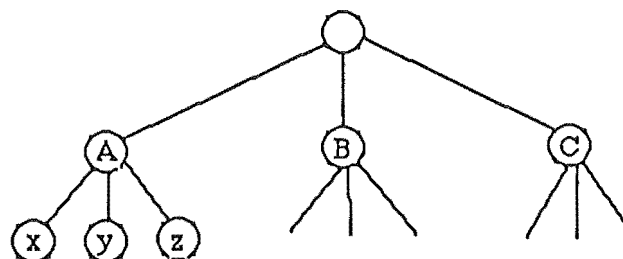
The second situation is when the user cannot understand the question, and it might occur quite frequently since it is not always possible to produce questions which everyone can always

understand fully. Sometimes it is more desirable to have reasonably brief questions which most people can understand most of the time, with special 'backup' questions for when a client answers "I don't know". The following QUESTION-declaration shows an example of this :-

```
(QUESTION msng-pgbrks-btwn-fls
  (YES-NO "Are there missing pagebreaks between the form-letters ?")
  (IF-DONT-KNOW
    (YES-NO "Is the first line of each form-letter being printed on the
      same page as the previous form-letter ?")
```

Alternatively, FA-rules could be written which have "(? factname)" in the premise condition.

Unfortunately this is not an adequate solution as it requires considerable extra work for the knowledge engineer. Some sort of automatic mechanism should also be provided so that if explicit 'dont-know' rules have not been produced, an "I don't know" response will still result in the system doing something sensible. A possible strategy for this could be based on the fact that a Mailmerge diagnosis is usually in (very roughly) a decision-tree form. If an "I don't know" answer is given then it is treated as meaning "no" until other branches of the tree have been considered. For example, suppose each node of the diagram in figure 7.9.1 represents a question in a diagnosis structure, with general questions at the top, more specific questions nearer the bottom, and leaf-nodes (not shown) being the final solutions.



If question A is answered with "I don't know", then "NO" will be assumed until B and C have been asked. If these do not lead to a solution then A is set to "YES", and x,y,and z are investigated. Unfortunately, the Mailmerge system is only based on a decision-tree structure very loosely, and hence this technique is not really applicable. However, a simple, and more general method has been developed which achieves essentially the same effect : Whenever an "I don't know" response is given, the fact-value is set to "?", and the fact-name is pushed onto a 'DONT-KNOW-stack'. The "?" value means that any rules which have been explicitly written to cope with the dont-know answer can still be fired, possibly resulting in a TRUE or FALSE value being assigned to the fact-name. Other rules will treat the "?" as meaning FALSE. If the system gets stuck (ie. no more rules can fire) then MERMEX will remove fact-names from the

DONT-KNOW-stack until one is found which still has the "?" value. The fact-value is then set to TRUE, which will hopefully enable the diagnosis to continue. To prevent confusion for the user, a message such as the following will be displayed :-

"Earlier you answered '?' when I asked you this question :

"Is a 'PA' being printed between each form-letter?"

What is your answer now ?

> ?

The following questions will assume that you answered "YES".

This method is quite simple, but very useful, since it avoids the necessity of having explicit rules to cope with every "I don't know" answer.

7.10 Recognising an Unsolvable Problem

When no more rules can fire, then this normally indicates that the problem cannot be solved. However, MERMEX will not give up quite so easily. Firstly, it will report any conflicting answers to the user, as stored in the CONFLICT-list. (Future versions of MERMEX might remove this step, for reasons given in Section 7.8). If no corrections can be made, then MERMEX will process any questions on the DONT-KNOW-stack, as discussed on the previous page. This may also enable the diagnosis to continue. The last resort is to perform the SUMMARY command, which will give the user the opportunity of checking and correcting any previous answers. If this fails as well, then MERMEX will admit defeat. The following is a summary of the above steps :-

- (1) Report and correct conflicting answers, as stored in the CONFLICT-list.
- (2) Process any questions on the DONT-KNOW-stack.
- (3) Perform the SUMMARY command which will ask the user to check that his previous answers were correct.
- (4) Give up.

To avoid step (4), the knowledge engineer should include weaker and weaker rules so that the system will not actually get stuck, but will just investigate less and less likely error-hypotheses, or re-investigate some earlier observation which might have been misinterpreted by the user. Some of these rules could explicitly cause a SUMMARY command to be executed, as for step (3) above. Obviously if the error is not in the knowledge-base then the problem will be unsolvable. However, if the error is known about then it is desirable for there to be several paths leading to that error being tested. This increases the chance that the system can solve a problem in a situation which was not directly considered by the knowledge engineer.

If a diagnosis is unsuccessful, it might be desirable to ask the user to enter an English description of the problem. This could be recorded, together with a history of the questions and responses given, to help the knowledge engineer maintain the knowledge base.

7.11 An Example Diagnosis (by MERMEX)

What type of problem are you having ?

- (1) Pagebreak problems
(eg. missing, extra, or incorrect pagebreaks)
- (2) Page-numbering problem
- (3) Page-sizes problem
- (4) Variable-data problem
(eg. names & addresses missing or incorrect)

1

How experienced are you at using Mailmerge ?

- (1) Novice
- (2) Intermediate
- (3) Expert

1

What type of pagebreak problem is it ?

- (1) Missing pagebreaks
- (2) Extra pagebreaks (eg. blank pages)
- (3) Incorrect pagebreaks

1

Is the missing pagebreak at a place where you have used a '.PA'?
YES

Is a 'PA' being printed at the position where the pagebreak is missing ?
NO

Are you printing form-letters ?
YES

Is the missing pagebreak occurring BETWEEN each form-letter ?
?

Is the top line of the second letter being printed
on the same page as the first letter?
YES

Load the form-letter file into Wordstar, and do a ^QC
to get to the end of it. OK?
YES

Is there a '.PA' at the end of the form-letter?
YES

Is a 'PA' being printed at the end of the last line
of the form-letter ?

SUMMARY

The following is a summary of what you have told me so far :

- (1) Pagebreaks are being missed out in the document.
- (2) A 'PA' is not being printed at the missing pagebreak
- (3) You are printing form-letters.
- (4) Pagebreaks are missing between each form-letter.
- (5) There is not a missing .PA at the end of the form-letter.

Which of these are incorrect ?

(enter 0 if none)

0

Is a 'PA' being printed at the end of the last line
of the form-letter ?

YES

Are you inserting a file just before the '.PA' ?

?

Is there a '.FI' command just above the '.PA'?

WHY?

If there is an inserted-file above the .PA then it may
have something wrong at the end of it (such as a missing
carriage-return, or trailing blanks).

Is there a '.FI' command just above the '.PA'?

YES

Load the inserted-file into wordstar, and do a ^QC
to get to the end of it. OK?

YES

Is the cursor directly below the last line of text ?

NO

SOLUTION :

You have missed out the carriage-return at the end of the
file. To insert a carriage-return, press <ENTER> after
doing a ^QC. This should solve your problem.

8. Evaluation of MERMEX

MERMEX is only at the prototype stage, and there is a great deal of further development still to be done. However, I am confident that the basic structure of the system would enable a successful working system to be produced. The current system only contains a very small amount of knowledge, and so the next step would be to build up the knowledge base sufficiently for it to be tested thoroughly on real-life problems. One of the most important lessons learnt from this project is how difficult and time-consuming the knowledge-acquisition process is. It is only through experience that this frequently stated fact really becomes clear. The amount of time available for this project has meant that only a fraction of the domain knowledge has been developed. This has partly been due to the fact that during most of the project I have been essentially acting as my own expert, knowledge engineer, and Mailmerge user, which is not a particularly productive approach. As a consequence, testing and evaluation of the system has been somewhat limited.

In section 7.11 a trace of a diagnosis performed by MERMEX was given, which illustrated some of the features provided. Producing the rules which were used in this diagnosis highlighted a number of aspects of the current design which should be modified or extended. One example is the functions provided for asking questions. These were found to be rather awkward in some situations, especially when whole groups of questions (as in a questionnaire) were required. However, this should only involve a few minor changes and additions to the current set of functions to make them more flexible.

Another feature which may need extending is the WHY command, which displays the current hypothesis being investigated. The main problem with this is that every rule needs to specify a hypothesis for the WHY facility to be effective. However, as discussed earlier, this may be a desirable condition for ensuring that the knowledge base is reasonably 'documented'.

The few tests performed so far have not been sufficient for determining whether this explanation facility would really be adequate in a working system. One difficulty arises when the right hand side actions of an FA-rule consist of a whole group of questions. The hypothesis description then needs to be worded so that a WHY response to any of these questions would give a meaningful explanation. Although several modifications still need to be investigated, the basic idea of associating an explanation message with each rule seems to be quite reasonable.

The SUMMARY feature certainly needs an extension to enable parts of the summary to be suppressed. A discussion on this was given in section 7.8.

There has not been very much work on the user-interface of MERMEX. Interaction is currently

restricted to Yes/No or Menu type questions. These are quite acceptable in most situations, but it might be desirable to implement a simple English language interface for entering the initial problem description. This could avoid some of the rather tedious initial questions in each diagnosis. This could also be used to enable clients to volunteer additional information. However, answering questions in English should be an optional feature, and the system should not ask questions such as "Describe ...". This would not only be difficult to implement well, but would also require too much 'effort' for the user. Yes/no questions are usually much easier to answer.

Many other systems allow facts to be arbitrary patterns, but in MERMEX they are restricted to being a name and a value. Although this may seem to limit flexibility, the advantages of the simplicity seem (so far) to outweigh any disadvantages. Therefore, it is not likely that future versions of MERMEX would change this. However, the introduction of variables to represent fact-values in the premise of rules might be desirable. Allowing lisp expressions on the left hand side might be another possible extension (similar to the 'TEST' clause in YAPS). This could give a lot more flexibility, and would enable, for example, probabilities to be used in rules. For example, rules could contain premise conditions such as "(> factname 0.8)". Unfortunately this introduces innumerable complications, and would need to be somewhat restricted if current features (such as consistency checking) are to remain compatible. It is unlikely, however, that numerical probabilities would be introduced in future versions. As emphasised several times in this report, numerical values tend to be quite meaningless in most situations. In a domain where the order of questioning is not particularly important, or where other control mechanisms are also included, or where rules are sufficiently precise for numerical weightings to be meaningful, then perhaps these would be appropriate. In the Mailmerge system this is not the case, and uncertainties and priorities seem to be much more effectively implemented with the rule-ordering technique.

However, the current rule-ordering technique might run into problems when the knowledge base becomes large. For example, if two groups of rules are in two widely separated parts of the knowledge base, then controlling the priority of these two groups can be awkward. However, rules should not be written in such a way that this would matter. One modification which will definitely be introduced is to allow a whole group of rules to be enclosed by a premise condition. This would allow a kind of hierarchical, nested structure, which could make controlling rule-priorities more convenient. If the priorities of two groups needs to be swapped, an extra condition can be put in the two enclosing 'group-premises' to achieve this. A more thorough evaluation of the rule-ordering technique can only really be done by developing a much larger knowledge-base, and finding (during the knowledge-acquisition process) the advantages and disadvantages.

Section 7.6 discussed how the facts NOVICE, INTERMEDIATE, and EXPERT can be used to allow for different types of user. However, it is doubtful whether it would be worth using the level of experience, since it effectively triples the amount of work required by the knowledge-engineer. It might be more sensible just to assume one particular type of user. Alternatively, the types of questions asked could be determined by the nature of the problem itself. In other words, questions which are related to a simple error such as a missing 'PA' could be written for a NOVICE user, based on the assumption that it is more likely that novice users would have that problem.

Techniques for knowledge acquisition is another area which I have not explored very much yet. The system so far has been based largely on my own Mailmerge 'expertise', and has only involved a very small subset of the domain, namely pagebreak problems. The method of obtaining sample diagnoses does obtain a reasonable amount of information, but requires an enormous amount of time and effort on the part of the expert. It would be more convenient to work with the expert in putting the knowledge directly into the production rules. The FA-rules provide quite a simple framework for collecting knowledge, since they are essentially just associations between observations and questions. The order of the rules could be ignored initially. Once the basic list of 'observations-to-questions' relationships have been produced, then the task of ordering the rules to indicate priorities, uncertainties, etc., can be performed.

As the knowledge base becomes larger, there will be an increasing need for some kind of tools to perform tasks such as keeping track of fact-names. Without these, it would be difficult to keep the knowledge-base manageable. Some sort of "Knowledge-base Development Environment" program would be a very useful facility.

A "Knowledge Compiler" could be an effective way to enable knowledge to be entered into the system in the most *meaningful* way (to a human), but then compiled into rules (or whatever) in the most *useful* way for the system. The 'reverse-rule' in the current version of MERMEX is actually an example of this.

The most important part of evaluating MERMEX is in determining whether the basic underlying knowledge structure is actually suitable. At this stage, the forward-chaining FA-rule structure seems reasonably effective. Putting knowledge into the system is quite 'easy', and there is a reasonable amount of flexibility. Most of the rules are essentially defining a hierarchy of observations, and this gives the system a reasonably structured framework for acquiring knowledge from the expert. The simple rule-ordering control mechanism makes the behaviour of the system quite predictable, and controllable, and hence maintaining a coherent dialogue is possible. The FF-rule structure also enables consistency-checking to be performed, and "Dont-know" answers can be catered for quite effectively.

Overall, the MERMEX design seems to be quite suitable for the Mailmerge problem, although, as said earlier, this can only really be determined after extensive testing has been performed, with real problems, real users, and a much more substantial knowledge base.

9. Conclusion

Although the "knowledge-acquisition bottleneck" has prevented the completion of a working system, progress made so far has been quite successful. There is still a great deal of work still to be done, but the basic underlying structure of the MERMEX design seems to be reasonably acceptable for the Mailmerge problem. After a few modifications and extensions have been made, I am confident that MERMEX could be used to build a reasonably powerful Mailmerge diagnosis system.

References

- [1] A. Barr, B. Feigenbaum; "The Handbook of Artificial Intelligence"
(includes an overview of MYCIN, INTERNIST, PIP, & EXPERT)
Pitman, 1982
- [2] J. Alty, M. Coombs; "Expert Systems - Concepts and Examples"
NCC Publications, 1984
- [3] F. Hayes-Roth,
D.A. Waterman,
D.B. Lenat; "Building Expert Systems"
Addison-Wesley, 1983
- [4] G. Kahn, J. McDermott; "The Mud System"
IEEE Expert, Spring 1986
- [5] S. Weiss, C. Kulikowski; "A Practical Guide to Designing Expert Systems"
Rowman & Allanheld 1984
- [6] A. Stefik, J. Aikens, R. Balzer, J. Benoit,
L. Birnbaum, F. Hayes-Roth, E. Sacerdoti;
"The Organisation of Expert Systems :
A Prescriptive Approach"
Xerox, 1984
- [7] R.H. Kemp, W. Teahan; "Frames as an Alternative to Production Rules"
International Conference on Future Advances in Computing. 1986
- [8] M. Schor; "Declarative Knowledge Programming :
Better than Procedural"
IEEE Expert, Spring 1986
- [9] T. O'Shea, M. Eisenstadt; "Artificial Intelligence :
Tools, Techniques, and Applications"
(includes "A Rational Reconstruction of the
MYCIN Consultation System", Cendrowska & Bramer)
Harper & Row, 1984
- [10] O.H. Winston; "Artificial Intelligence"
Addison-Wesley, 1977
- [11] D. Michie; "Introductory Readings in Expert Systems"
- [12] R. Forsyth; "Expert Systems - Principles and Case Studies"
Chapman & Hall, 1984
- [13] D. Huntingdon; "EXSYS (Expert System Development) User's Manual"
(not to be confused with Adata!)
1984
- [14] M. Neale; "Expert Systems in Prolog"
University of Canterbury, Honours Project, 1984
- [15] R. Wilensky; "Lispcraft"
W.W. Norton & Co., 1984
- [16] Wordstar Reference Manual (includes Mailmerge)
Micropro Corporation

- [17] E.M. Allen "YAPS : Yet Another Production System"
University of Maryland, A.I. Group, 1983
 - [18] A. Zissos "Generating Advice by Monitoring User Behaviour"
(describes the "Anchises" system)
M.Sc. Report, University of Calgary, 1985
 - [19] C. Forgy, "OPS5 User's Manual"
Carnegie Mellon University, 1981
 - [20] L. Brownston, R. Farrel,
E. Kant, N. Martin
"Programming Expert Systems in OPS5
- An Introduction to Rule-Based Programming."
Addison-Wesley, 1985
-

Appendix A

Why I Did Not Use YAPS

My first 'working' Expert System was written in YAPS ("Yet Another Production System") [17], and was based on the diagnosis structure discussed in Section 6 (see figures 6.3 and 6.4). YAPS was a very good place for starting the development of a simple prototype system, since it removed the need for writing my own production system. However, I found that there were a number of aspects of YAPS which dissuaded me from using it for further development.

The easiest way to show why I found YAPS undesirable for the Mailmerge system is by comparing a rule from MERMEX with a roughly equivalent YAPS rule. The MERMEX rule is as follows :-

```
(FF (msgng-pgbrk-btwn-fls)
    (PA-printed-eoln)      --->    (PA-printed-end-fl))
```

Firing this rule will result in the right hand side fact being asserted, and marked as 'INFERRED', a consistency check is performed, and several other control functions are executed, such as marking the rule as 'FIRED'. MERMEX will also automatically generate the 'reverse-rule' (as explained in Section 8.2).

In YAPS, the rule might be something like :-

```
(P FF-rule-PA-printed-end-fl
  (in-FF-phase)
  (~ (FIRED FF-PA-printed-end-fl))
  (msgng-pgbrks-btwn-fls 'TRUE)
  (PA-printed-eoln 'TRUE)
  (~ (PA-printed-end-fl)) --->    (fact FIRED FF-PA-printed-end-fl
                                   (fact PA-printed-end-fl 'TRUE)
                                   (fact PA-printed-end-fl 'INFERRED))
```

Instead of the control information being kept in some general procedure, much of it has to be explicitly stated in each rule. This makes each rule much less clear, and makes changing the control strategy very difficult. The MERMEX rule can essentially be interpreted in whatever way is desirable (during development), since I can modify the control strategy whenever I want to. The YAPS rule, on the other hand, has to obey YAPS syntax, and can only be interpreted one way. For an initial system, using an existing program such as YAPS may be quite appropriate, but this is often not suitable when a variety of different control strategies are being investigated. The Expert System designer needs a much more flexible language (such as Lisp) in this situation. The Yaps

rule above does not even perform many of the functions of the MERMEX rule. For example, there are no consistency checks, and the 'reverse-rules' are not generated. Modifying the control strategy in a YAPS system would probably involve changing every rule.

One way of avoiding these problems would be to represent Mailmerge rules as YAPS facts. In other words, the rules would simply be patterns of symbols, possibly in the same form as a MERMEX rule. General-purpose YAPS rules could then act on these "facts", which are actually Mailmerge rules. In other words, YAPS is just being used as a programming language to produce the general control mechanism. However, if this approach is to be taken then the program may as well be written in Lisp, which is in fact what I did, hence resulting in the MERMEX system.

To summarise the above : if Mailmerge rules are encoded directly into YAPS rules, then they become awkward, inflexible, and messy, since each rule must contain a substantial amount of control information. This approach is treating YAPS as a kind of limited and cumbersome Expert System 'shell'. If, on the other hand, the Mailmerge rules are treated as YAPS facts, so that general-purpose YAPS rules can process them, then there is not much advantage in using YAPS. This latter approach is just using YAPS as a general-purpose programming language.

The "most-recent-fact-bindings" strategy used in YAPS to determine rule-priority was also found to be undesirable for the Mailmerge problem, as was discussed in section 8.3. One of the earlier versions of the system had several classes of rules, each of which was fired during a different phase of operation (similar to FF and FA rules). Doing this in YAPS required every rule to have a special condition clause to prevent it from firing except in the correct phase. Controlling priorities of rules was even more awkward. Also, since a rule is re-fired whenever any of the left hand side facts gets re-asserted, it was necessary to add yet another condition in each rule to check whether the rule has already fired earlier.

Implementing certainty-values (if these were necessary) would also be difficult, since they would be represented by patterns of symbols, such as "(fact-x 0.4)". Calculating or modifying these values would require removing the old pattern, and asserting a new one. It would be necessary to do this in the context of a YAPS rule, since facts cannot be accessed from within a user-defined lisp function.

In the MERMEX system, consistency checks are performed, "I don't know" answers are automatically catered for, a summary of the information obtained from the user can be recorded and displayed, and so on. If I had continued to use YAPS, none of these features would have been developed. Nevertheless, YAPS was certainly a good place to start, even just from the point of view becoming familiar with an existing production system. YAPS may be suitable for some domains, since it is very similar to OPS5 which has been used to produce the 'R1' VAX-configuration system [2], the MUD system discussed in Section 5, and many other

applications. However, for the Mailmerge problem it was found to be an inappropriate tool for developing the required knowledge representations and control mechanisms.

Appendix-B

The MERMEX Reference Manual

Formal Definition of Terms

<fact-name>	::-	any lisp symbol
<value>	::-	any lisp value
<fact>	::-	(<fact-name>) (NOT <fact-name>) (? <fact-name>) (<fact-name> <value>) (NOT <fact-name> <value>)
<facts>	::-	<fact> <fact> ...
<fact-exprs>	::-	<fact-expr> <fact-expr> ...
<fact-expr>	::-	<fact> (AND <fact-exprs>) (OR <fact-exprs>) (NOT <fact-exprs>)
<actions>	::-	<action> <action> ...
<action>	::-	any lisp expression
<message>	::-	<message-item> <message-item> ...
<message-item>	::-	"...." (the string is printed as it appears) <fact-name> (the fact's DESCription is printed) (<fact-name>) (the value of the fact is printed) N (a linefeed is printed)

Action Evaluation Functions

(AND <actions>)

- Performs the sequence of actions until one returns a nil value.
- Returns t if all of the actions are performed.

(OR <actions>)

- Performs the sequence of actions until one returns a non-nil value.
- Returns t if any of the actions returns a non-nil value.

(NOT <actions>)

- Performs the sequence of actions until one returns a non-nil value.
- Returns nil if any of the actions returns a non-nil value.

(DO <actions>)

- Performs the sequence of actions until one returns a nil value.
- Returns t.

(ALL <actions>)

- Performs all of the actions.
- Returns t.

(IF-YES <actions>)

- If the most recent YES-NO question was answered with a "YES", then performs the sequence of actions as for "AND".
- Returns t.

(IF-NO <actions>)

- Similar to IF-YES

(IF-DONT-KNOW <actions>)

- Similar to IF-YES

Fact Evaluation Functions

(IF (<fact-exprs>) <actions>)

- If all of the fact-expressions are true, then perform an AND of the actions.
- Returns t.

(TEST <fact-exprs>)

- returns t if all of the fact-expressions are true.

(ASSUMES <facts>)

- returns t if all of the facts are either true or have "?" values.

(NEW-FACTS <facts>)

- asserts the specified facts.

(NF <fact-name>)

- asserts the fact '(<fact-name>)'.

(YES-FACTS <facts>)

- asserts the facts, iff the most recent YES-NO answer was a "YES".

(NO-FACTS <facts>)

- asserts the facts, iff the most recent YES-NO answer was a "NO".

(YES-NO-FACTS <facts>)

- If the most recent YES-NO answer was a "YES" then the facts are asserted.

- If the answer was a "NO" then assert the opposite of the facts.

(VALUE <fact-name>)

- returns the value of the fact, or nil if it is "out-of-date". This function enables fact-values to be accessed from within lisp. Lisp code can access facts directly by their name, but this will not check their 'version-number'.

Knowledge Definition Functions

(FF <fact-exprs> <facts>)

- defines an FF-rule.

(FF* <fact-exprs> <facts>)

- as for FF, but reverse-rules are not created.

(FA [<priority>] <fact-exprs> <actions>)

- defines an FA-rule.

- the optional <priority> is either 0,1,or 2.

(DESC <fact-name> <message>)

- defines a description of a fact-name.

(QUESTION <fact-name> <actions>)

- (or just 'Q')

- defines the actions to be performed when a
"(QUESTION <fact-name>)"
command appears on the right-hand-side of an FA-rule.


```
( MENU <fact-name>
  ( MSG <message> )
  ( OPTION <fact-name>
    ( MSG <message> )
    <actions>
  )
  ( OPTION <fact-name>
    ...
  )
  ...
)
```

- defines a menu to be displayed when a "(MENU <fact-name>)" command appears on the right-hand-side of an FA-rule.

```
( HYPOTHESIS <fact-name> <message> )
```

- defines a hypothesis-message.
- see DESC for a definition of <message>

```
( SOLUTION <fact-name> <actions> )
```

- defines the list of actions to be performed when a "(SOLUTION <fact-name>)" command appears on the right-hand-side of an FA-rule.
- MERMEX will terminate after the actions have been executed.

Asking Questions

```
( QUESTION <fact-name> )
```

- Performs the actions defined by the QUESTION-declaration (see "Knowledge Definition Functions").
- Returns t if the <fact-name> has the value TRUE when the QUESTION command is completed.
- The sequence of actions is not performed if the <fact-name> already has a non-nil value, or if the QUESTION command was done earlier.

```
( MENU <fact-name> )
```

- Performs the actions defined by the QUESTION-declaration (see "Knowledge Definition Functions").

```
( YES-NO <fact-name> <message> )
```

- displays the message (as defined in DESC).
- waits for a "YES", "NO" or "?" response, and sets the value of the fact-name to TRUE, FALSE, or ?, respectively.

```
( YES-NO <message> )
```

- this can only be used within a QUESTION declaration.
- the <fact-name> of the QUESTION is used.

```
( NO-YES ... )
```

- as for the two YES-NO functions above, except that if a "YES" answer is given, then the fact-value is set to FALSE, (and TRUE for "NO").

(ENTER-VALUE <fact-name> <message>)

- Prints the message, and waits for a value to be entered, which is then assigned to the fact-name.

Explanation Functions

(SUMMARY)

- displays the description of the facts whose values have been obtained by asking the user.
- the user is then given the opportunity to correct any of these answers.

(STUCK-SUMMARY)

- as for SUMMARY, except that the message "I can't think of a solution .." is printed first.

(HYPOTHESIS <fact-name>)

- defines the current hypothesis as being the message given in the fact-name's HYPOTHESIS definition.

(HYPOTHESIS <message>)

- defines the current hypothesis to be the specified message.

(HYPOTHESIS)

- displays the current hypothesis.

(MSG <arguments>)

- this is identical to the Maryland Extensions "msg" function.

(SOLUTION <fact-name>)

- performs the actions defined in a SOLUTION-declaration (see "Knowledge Definition functions"), and then terminates the diagnosis.

```

(FA (PA-printed-indented)      -->      (Q PA-after-mf)
                                           (Q* look-end-mf)
                                           (Q blns-end-mf)
                                           (SOLUTION blanks-end-mf))

(FA (PA-printed-at-msg-pgbrk)
    (NOT PA-indented)
    (NOT incorrect-PA-printed) --> (Q suppressed-page-formatting))

(FA (PA-printed-at-msg-pgbrk)
    (NOT PA-indented)
    (NOT incorrect-PA-printed)  -->      (Q* look-PA-printed)
                                           (Q incorrect-PA))

(FA (PA-printed-at-msg-pgbrk)
    (NOT incorrect-PA)          -->      (STUCK-SUMMARY))

;----- msg-pgbrks-btwn-fls

(FA (check-PA)                  -->      (OR (Q missing-PA)
                                           (Q incorrect-PA)))

(FA (check-PA-end-fl)          -->      (Q* look-PA-end-fl)
                                           (NF check-PA))

(FA (check-PA-printed)         -->      (OR (Q PA-printed)
                                           (Q PA-printed-eoln)
                                           (Q
PA-printed-start-of-line)))

(FA (check-PA-printed-btwn-fls) -->      (Q* look-end-fl)
                                           (NF check-PA-printed))

(FA (msg-pgbrks-btwn-fls)
    (mf-end-fl)                 -->      (Q* look-end-mf))

;-----

(FA (msg-pgbrks-btwn-fls)      -->      (Q msg-PA-end-fl))

(FA (msg-pgbrks-btwn-fls)      -->      (Q PA-printed-eoln-fl))

(FA (msg-pgbrks-btwn-fls)      -->      (NF
check-PA-printed-btwn-fls))

(FA (msg-pgbrks-btwn-fls)      -->      (Q mf-end-fl))

(FA (msg-pgbrks-btwn-fls)      -->      (Q incorrect-PL))

(FA (msg-pgbrks-btwn-fls)      -->      (SUMMARY))

;----- top level rules

(FA (start)                    -->      (MENU problem-menu)
                                           (MENU experience-menu))

(FA (pgbrks)                   -->      (MENU pgbrks-menu))

(FA (msg-pgbrks)               -->      (Q msg-pgbrk-where-used-PA))

(FA (msg-pgbrk-where-used-PA)  -->      (Q PA-printed-at-msg-pgbrk))

(FA (msg-pgbrks)               -->      (Q form-letters)
                                           (WHY
    "If pagebreaks are missing between form-letters"
    "then you" N

```

```

"may have an incorrect (or missing) '.PA' "
"at the end of the file.")
(Q msgng-pgbrks-btwn-fls))

(FA (msgng-pgbrks)
  (NOT msgng-pgbrk-where-used-PA) --> (MENU why-msgng-pgbrk))

;-----

;===== FF-rules

(FF (PA-printed-eoln-fl) --> (PA-printed-eoln))
(FF (OR (PA-printed-eoln)
  (PA-printed-separate-line)
  (PA-printed-start-of-line)) --> (PA-printed))

;===== SOLUTIONS

(SOLUTION missing-dot-before-PA
  (msg "You have missed out the dot before the 'PA'." N
    "This is causing the incorrect pagebreaks." N))

(SOLUTION msgng-crgret
  (msg "You have missed out the carriage-return at the end
    of the" N
    "file. To insert a carriage-return, press <ENTER>
    after " N
    "doing a ^QC. This should solve your problem."))

;===== DEScriptions

(DESC form-letters
  "You are[/ not] printing form-letters.")
(DESC pgbrks
  "Pagebreaks are[/ not] being missed out in the document.")
(DESC msgng-pgbrks-btwn-fls
  "Pagebreaks are[/ not] missing between each form-letter.")
(DESC msgng-pgbrks-where-used-PA
  "The missing pagebreak is[/ not] at a '.PA'")
(DESC PA-printed-at-msgng-pgbrk
  "A 'PA' is[/ not] being printed at the missing pagebreak")
(DESC msgng-PA-end-fl
  "There is[/ not] a missing .PA at the end of the
  form-letter.")
(DESC PA-after-MF
  "You are inserting a file (with .FI) just above the
  '.PA'.")
(DESC PA-printed-eoln-fl
  "There is a 'PA' being printed at the end of the" N
  "last line of the form-letter.")

;===== QUESTIONS

;----- top level

(QUESTION problem
  (YES-NO "Hello! Are you having a problem ?")
)

(MENU problem-menu

```

```

(MSG "What type of problem are you having ?")
(OPTION pgbrks
  (MSG "Pagebreak problems" N
    (C TAB) " (eg. missing, extra, or incorrect
              pagebreaks)"))
(OPTION page-numbering
  (MSG "Page-numbering problem"))
(OPTION page-sizes
  (MSG "Page-sizes problem"))
(OPTION vardata
  (MSG "Variable-data problem" N
    (C TAB) " (eg. names & addresses missing or
              incorrect)"))
)

(MENU experience-menu
  (MSG "How experienced are you at using Mailmerge ?")
  (OPTION NOVICE
    (MSG "Novice"))
  (OPTION INTERMEDIATE
    (MSG "Intermediate"))
  (OPTION EXPERT
    (MSG "Expert"))
)

(MENU pgbrks-menu
  (MSG "What type of pagebreak problem is it ?")
  (OPTION msgng-pgbrks
    (MSG "Missing pagebreaks"))
  (OPTION extra-pgbrks
    (MSG "Extra pagebreaks (eg. blank pages)"))
  (OPTION incorrect-pgbrks
    (MSG "Incorrect pagebreaks"))
)

(QUESTION msgng-pgbrks
  (YES-NO "Are pagebreaks being missed out ?"))

(QUESTION extra-pgbrks
  (YES-NO "Are extra blank pages being printed ?"))

(QUESTION wrong-pgbrks
  (YES-NO "Are pagebreaks being printed in unexpected
           positions ?"))

(QUESTION form-letters
  (YES-NO "Are you printing form-letters ?")
  (IF-DONT-KNOW
    (YES-NO "Are you using the .DF and .RV commands?"))
)

(QUESTION suppressed-page-formatting
  (YES-NO "Did you answer 'YES' to the SUPPRESS PAGE
           FORMATTING option?")
  (IF-YES
    (SOLUTION suppressed-page-formatting))
)

;----- missing pagebreaks

(QUESTION msgng-pgbrks-btwn-fls
  (YES-NO "Is the missing pagebreak occurring BETWEEN each
           form-letter ?")
  (IF-DONT-KNOW
    (YES-NO "Is the top line of the second letter being
           printed" N

```

```

        "on the same page as the first letter?"))
    )

(QUESTION msgng-pgbrk-where-used-PA
  (YES-NO
    "Is the missing pagebreak at a place where you have
    used a '.PA' ?"))

(QUESTION msgng-pgbrk-after-mf
  (YES-NO "Is the missing pagebreak at the end of an" N
    "inserted (.FI) file ?"))

(QUESTION msgng-PA-end-fl
  (Q* look-end-fl)
  ; (MSG "HERE" N)
  (NO-YES "Is there a '.PA' at the end of the form-letter?")
  (IF-NO
    (SOLUTION msgng-PA))
  )

(QUESTION incorrect-PA-end-fl
  (Q* look-PA-end-fl)
  (Q incorrect-PA)
  (QVALUE 'TRUE)
  )

(QUESTION msgng-pgbrk-after-mf
  (YES-NO "missing pagebreak after merged file ?"))

(QUESTION PA-after-mf
  (YES-NO "Are you inserting a file just before the '.PA' ?")
  (IF-DONT-KNOW
    (YES-NO "Is there a '.FI' command just above the '.PA' ?"))
  )

;----- PA printed

(QUESTION PA-printed-at-msgng-pgbrk
  (YES-NO
    "Is a 'PA' being printed at the position where the pagebreak
    is missing ?"))

(QUESTION PA-printed-btwn-fls
  (YES-NO "Is a 'PA' being printed between each form-letter ?")
  (IF-NO
    (QUESTION PA-printed-eoln-fl))
  )

(QUESTION PA-printed-indented
  (YES-NO "Does the 'PA' have blanks on the left of it ?"))

(QUESTION PA-printed-eoln
  (YES-NO "Is a 'PA' being printed at the end of the last
    line of text?"))

(QUESTION PA-printed-separate-line
  (YES-NO "Is a 'PA' being printed on a separate line ?"))

(QUESTION PA-printed-start-of-line
  (YES-NO "Is a 'PA' being printed at the start of the
    line of text ?"))

(QUESTION PA-printed-eoln-fl
  (YES-NO "Is a 'PA' being printed at the end of the last line" N
    "of the form-letter ?"))

```

```

(QUESTION PA-printed-btwn-fls
  (YES-NO "Is a 'PA' being printed between each form-letter?"))

(QUESTION incorrect-PA-printed
  (NO-YES "Is there a dot in front of the 'PA' being printed?")
  (IF-YES
    (YES-NO "Are there blanks between the dot and the PA?"))
  )

;----- incorrect PA

(QUESTION incorrect-PA
  (OR (Q missing-dot)
    (Q blanks-before-dot)
    (Q blanks-after-dot)
    (Q qmark-rhs-screen)
  )
  (QVALUE 'TRUE)
  )

(QUESTION incorrect-PA-at-msg-pgbrk
  (YES-NO "Go into Wordstar and find the 'PA' which was
    printed." N)
  (IF-YES (QUESTION incorrect-PA))
  )

(QUESTION incorrect-PA-end-fl
  (YES-NO "Go into Wordstar and find end of the form-letter." N
    "Is there a 'PA' there ?")
  (IF-YES (QUESTION incorrect-PA))
  )

(QUESTION missing-dot
  (YES-NO "Is there a '.' in front of the PA ?")
  (IF-NO (SOLUTION missing-dot))
  (IF-YES (QUESTION blank-before-dot))
  )

(QUESTION blank-before-dot
  (YES-NO "Are there any blanks in front of the '.' ?")
  (IF-YES (SOLUTION blank-before-dot))
  (IF-NO (QUESTION qmark-dot-command))
  )

(QUESTION qmark-dot-command
  (YES-NO "Is there a question-mark printed on the right-hand
    side" N
    "of the line ?")
  (IF-YES (SOLUTION bad-dot-command))
  )

(QUESTION PA-indented
  (YES-NO "Does the .PA have blanks on the left of it?")
  (IF-YES
    (SOLUTION blanks-before-PA))
  (IF (NOVICE)
    (YES-NO "Do a ^QS. Is the cursor on top of the dot?")
    (IF-YES
      (SOLUTION blanks-before-PA))
    )
  )
  )

;----- looking at files

(QUESTION look-PA-in-file

```



```

    (IF (NOT (looking-at PA))
      (YES-NO "Load in the file containing the .PA. and" N
        "find the PA on the screen.  OK?")
      (NEW-FACTS (looking-at PA))
    )
    (MSG "here" N)
    (QVALUE 'TRUE)
  )

  (QUESTION look-PA-printed
    (IF (NOT (looking-at PA))
      (YES-NO "Load the file which contains the PA which
        was printed. OK?")
      (IF-YES
        (NEW-FACTS (looking-at PA)))
      )
    )

  (QUESTION look-end-fl
    (IF (NOT (looking-at end-fl))
      (YES-NO "Load the form-letter file into Wordstar, and
        do a ^QC" N
        "to get to the end of it.  OK?")

      (IF-NO
        : (MSG "in IF-NO" N)
        (Q look-end-fl))
        : (MSG "HERE-3" N)
        (NEW-FACTS (looking-at end-fl))
      )
      : (MSG "at look-end-fl" N)
      (QVALUE 'TRUE)
    )

  (QUESTION look-end-mf
    (IF (NOT (looking-at end-mf))
      (YES-NO "Load the inserted-file into wordstar, and do
        a ^QC" N
        "to get to the end of it.  OK?")

      (IF-NO
        (Q look-end-mf))
        (NEW-FACTS (looking-at end-mf))
      )
      (QVALUE 'TRUE)
    )

  ;----- trailing blanks
  ; carriage returns, etc.

  (QUESTION blanks-end-mf
    (Q look-end-mf)
    (YES-NO "After doing a ^QC.  are there blanks on the left" N
      "of the cursor?")
    (IF-YES
      (SOLUTION trailing-blanks))
    (IF (NOT EXPERT)
      (YES-NO "Try doing a ^S.  Does the cursor move to the end
        of" N
        "the previous line?")
      (IF-NO
        (SOLUTION trailing-blanks))
      )
    (QVALUE 'FALSE)
  )

  (QUESTION msg-crgret

```

```
(NO-YES "Is the cursor directly below the last line of text ?")
(IF-NO
  (SOLUTION msg-crgret))
)
```

```
(QUESTION incorrect-PL
  (NO-YES "Are you using the correct pagelength (.PL) ?")
  (IF-NO
    (SOLUTION incorrect-PL))
)
```

```
===== load
```

```
(load 'ver3a.1)
(load 'ver3b.1)
(load 'ver3l.1)
```

```
===== run
```

```
(defun run fexpr (files)
  (setq trace nil
        else t)
  (if (member (car files) '(a all l))
      (setq files '(FF.1 FA.1 Q.1 SOLN.1 DESC.1)))
  (if files
      (my-eval 'LOAD-FILES files))
  (setq VERSION (newsym 'V)
        FACT-VERSION (newsym 'V)
        FF-VERSION (newsym 'V))
  (setq %LAST-YES-NO nil
        %SUMMARY-list nil
        %HISTORY nil
        %CONFLICT-list nil
        %DONT-KNOW-list nil
        %ASKED-list nil
        %HYPOTHESIS nil
        %LAST-HYPOTHESES nil)
  (array %FA-firelist t 10)
  (store (%FA-firelist 0) nil)
  (store (%FA-firelist 1) nil)
  (store (%FA-firelist 2) nil)
  (NEW-FACTS (start))
  (solve)
)
```

```
===== solve
```

```
(defun solve ()
  (prog ()
    (msg N "-----" N N)
    keep-going
    (for while (fire-FA-rule))
    (if (not (STUCK))
        (go keep-going))
    (UNSOLVABLE)
  )
)
```

```
===== fire-FA-rule
```

```
(defun fire-FA-rule ()
  (prog (FA-name priority)
    (if (setq priority (cond ((setq FA-name (car (%FA-firelist
0))) 0)
                           ((setq FA-name (car (%FA-firelist
1))) 1)
                           ((setq FA-name (car (%FA-firelist
2))) 2)))
        (store (%FA-firelist priority) (cdr (%FA-firelist
priority)))
        (putprop FA-name VERSION 'DONE)
        (if trace
            (msg N " -- firing " FA-name " (priority=" priority
)" N))
        (my-eval 'DO (get FA-name 'RHS))
        (return t)
  )
)
```

```

        else
            (return nil)
        )
    )
)

;===== next-FA

(defun next-FA ()
  (prog (FA-name)
    (for let (priority -1)
      while (and (< priority 9)
                 (not FA-name))
        do
          (setq priority (+ 1 priority))
          (setq FA-name (car (%FA-firelist priority)))
        )
    )
  )

;===== STUCK

      ;-- only stuck if "correct-conflicts" etc. do not
      ;-- fire any FA-rules

(defun STUCK ()
  (not (and (cond
             ((correct-conflicts))
             ((process-DONT-KNOWs))
             ((STUCK-SUMMARY)))
            ((next-FA))
          )
    )
)

;===== STUCK-SUMMARY

(defun STUCK-SUMMARY ()
  (msg N "I'm having trouble finding of a solution." N)
  (SUMMARY)
)

;===== process-DONT-KNOWs

(defun process-DONT-KNOWs ()
  (setq DK-list %DONT-KNOW-list)
  (for let (found nil)
    f in DK-list
    until found
    do
      ;-- remove facts from DONT-KNOW-list
      (setq %DONT-KNOW-list (cdr %DONT-KNOW-list))
      (if (eq (fact-value f) '?')
        (msg N "Earlier you answered '?' to the following
question : " N)
        (my-eval 'msg (get f 'YES-NO)) (msg N N)
        (YES-NO q "What is your answer now ?")
        (cond
          ((eq q 1.0) (my-eval 'NEW-ASKED-FACTS `((.f)))
                     (setq found t))
          ((eq q '?') (my-eval 'NEW-ASKED-FACTS `((.f)))
                     (setq found t)
                     (msg N "The next few questions will
assume")
                     (msg "that you answered 'YES'." N N)
                     )
        )
      )
  )
)

```

```
)  
    finally  
      found  
    )  
  )  
  
;===== UNSOLVABLE  
  
(defun UNSOLVABLE ()  
  (msg N "I cannot solve your problem." N N)  
  (print-HYPOTHESIS)  
)  
  
;===== correct-conflicts  
  
(defun correct-conflicts ()  
  (prog (conflict-list asked-list)  
    (cond  
      ((null %CONFLICT-list)  
        nil)  
      (else  
        (setq conflict-list %CONFLICT-list)  
        (setq %CONFLICT-list nil)  
        (msg N "Some of your earlier answers seem to conflict :\" N)  
        (for symbol in %CONFLICT-list  
          bind (count 1 (add1 count))  
          do  
            (msg "(" count ") "  
            (print-DESC symbol) (msg n)  
            )  
        (msg N "Which of these are incorrect (0 for none) ?")  
        (input-choices (length %CONFLICT-list))  
        (cond  
          ((and (eq (car YES-choices) 0)  
                (null (cdr YES-choices)))   nil)  
          (else  
            (setq change-list nil)  
            (for n in YES-choices  
              do  
                (push (nth (1- n) %CONFLICT-list) change-list)  
              )  
            ;-- clears facts and 'ASKEDs  
            (setq FACT-VERSION (newsym 'V))  
            ;-- clears FF-rules (DONE)  
            (setq FF-VERSION (newsym 'V))  
            (setq %SUMMARY-list nil)  
            (setq asked-list %ASKED-list)  
            (setq %ASKED-list nil)  
            (for fact in %asked-list  
              do  
                (if (member (fname fact) change-list)  
                    (NEW-ASKED-FACTS (list (not-fact fact)))  
                  else  
                    (NEW-ASKED-FACTS (list fact))  
                  )  
              )  
            )  
          )  
        t  
      )  
    )
```

```

;===== my-eval

(defmacro my-eval (action parameters)
  `(eval (cons ,action ,parameters)))
  )

;===== do-ACTION

(defun do-ACTION (action)
  (eval action)
  ;(if (fact? action)
  ;  (test-fact action)
  ;  else
  ;  (eval action)
  ;  )
  )

;===== AND

(defun AND fexpr (actions)
  (cond
    ((null actions))
    ((and (do-ACTION (car actions))
          (my-eval 'AND (cdr actions))))
  )
  )

;===== OR

(defun OR fexpr (actions)
  (cond
    ((null actions) nil)
    (or (do-ACTION (car actions))
        (my-eval 'OR (cdr actions)))
  )
  )

;===== NOT

(defun NOT fexpr (actions)
  (not (my-eval 'OR actions))
  )

;===== ALL

(defun ALL fexpr (actions)
  (mapc 'do-ACTION actions)
  t
  )

;===== DO

(defun DO fexpr (actions)
  (my-eval 'AND actions)
  t
  )

;===== AND-FACTS

(setq TEST 'AND-FACTS)
(defun AND-FACTS fexpr (fact-exprs)
  (cond
    ((null fact-exprs))
    ((and (test-fact-expr (car fact-exprs))
          (my-eval 'AND-FACTS (cdr fact-exprs))))
  )
  )

```

```

)

;===== OR-FACTS

(defun OR-FACTS fexpr (fact-exprs)
  (cond
    ((null fact-exprs) nil)
    ((or (test-fact-expr (car fact-exprs))
         (my-eval 'OR-FACTS (cdr fact-exprs))))
  )
)

;===== NOT-FACTS

(defun NOT-FACTS fexpr (fact-exprs)
  (not (my-eval 'OR-FACTS fact-exprs))
)

;===== ASSUMES

(defun ASSUMES fexpr (facts)
  (cond
    ((null facts))
    ((and (or (test-fact (car facts))
              (test-fact '(? ,(fname (car facts)))))
         (my-eval 'ASSUMES (cdr facts))
        ))
  )
)

;===== DESC

(defun DESC fexpr (args)
  (clear-facts (list (car args)))
  (putprop (car args) (cdr args) 'DESC)
  t
)

;----- print-DESC

(defun print-DESC (symbol)
  (let ((desc (get symbol 'DESC)))
    (if desc
        (eval (append (list 'MYMSG symbol) desc))
        else
        (msg "(")
        (msg (cond
              ((eq (eval symbol) 'TRUE)   "")
              ((eq (eval symbol) 'FALSE)  "NOT ")
              ((eq (eval symbol) '?)      "? ")
            )) symbol)
        )
  )
)

(defun MYMSG fexpr (symbol-message)
  (prog (m symbol message)
    (setq symbol (car symbol-message)
          message (cdr symbol-message))
    (for m in message
      do
        ;(msg "m=" m N)
        (cond
          ((eq m 'N) ;-- N
           (msg N)
          )
          ((symbolp m) ;-- factname
           )
        )
    )
  )
)

```

```

        (print-DESC m)
      )
      ((listp m)                                ;-- (factname)
        (msg (fact-value (car m)))
      )
      (else                                     ;-- "...[is/is
not]..."
        (msgstr symbol m)
      )
    )
  )
)

```

```

(defun msgstr (symbol m)
  (setq chrs (cdr (explode m)))
  (for let (count -1)
    while (< count (- (length chrs) 2))
    do
      (setq count (+ 1 count))
      (cond
        ((eq (nth count chrs) '\[)
          (setq count (+ 1 count))
          (selectq (fact-value symbol)
            (TRUE
              (msg N "at-TRUE" N)
              (for while (neq (nth count chrs) '/')
                do
                  (msg (nth count chrs))
                  (setq count (+ 1 count)))
              (for while (neq (nth count chrs) '\])
                do
                  (setq count (+ 1 count))
                )
            )
            (otherwise
              (msg N "at-FALSE" N)
              (for while (neq (nth count chrs) '/')
                do
                  (setq count (+ 1 count)))
              (setq count (+ 1 count))
              (for while (neq (nth count chrs) '\])
                do
                  (msg (nth count chrs))
                  (setq count (+ 1 count)))
            )
          )
      )
    )
  )
  (else
    (msg N "count=" count N)
    (msg (nth count chrs)))
  )
)
)

```

===== F

```

(defun NF fexpr (factname)
  (eval `(NEW-FACTS ,factname))
)

```

===== HYPOTHESIS

```

(setq WHY 'HYFOTHEISIS)
(defun HYPOTHEISIS fexpr (args)

```



```

(let ((head (car args))
      hyp)
  (cond
    ((null args)                                     ;-- (HYPOTHESIS)
      (print-HYPOTHESIS)
    )
    ((and (symbolp head)                             ;-- (HYPOTHESIS
symbol message)
      (cdr args))
      (setq hyp (cdr args))
      (putprop head hyp 'HYPOTHESIS)
    )
    ((symbolp head)                                   ;-- (HYPOTHESIS
symbol)
      (setq hyp (get head 'HYPOTHESIS))
    )
    (else                                             ;-- (HYPOTHESIS
message)
      (setq hyp args)
    )
  )
  (if hyp
    (setq %HYPOTHESIS hyp)
    (setq %LAST-HYPOTHESES (list hyp))
  )
)
)

```

```

;-----
(defun print-HYPOTHESIS ()
  (msg N "-----" N)
  (eval (append '(MYMSG RUB) %HYPOTHESIS))
  (msg N "-----" N)
)

```

===== IF-YES etc

```

(defun IF fexpr (premise-actions)
  (if (my-eval 'TEST (list (car premise-actions)))
    (my-eval 'DO (cdr premise-actions))
    )
  t
)

```

```

(defun IF-YES fexpr (actions)
  (if actions
    (if (eq %LAST-YES-NO 'Y)
      (my-eval 'DO actions))
    t
  else
    (eq %LAST-YES-NO 'Y)
  )
)

```

```

(defun IF-NO fexpr (actions)
  (if actions
    (if (eq %LAST-YES-NO 'N)
      (my-eval 'DO actions))
    t
  else
    (eq %LAST-YES-NO 'N)
  )
)

```

```
;-----  
(defun IF-DONT-KNOW fexpr (actions)  
  (if actions  
    (if (eq %LAST-YES-NO '?)  
      (my-eval 'DO actions)  
      t  
    else  
      (eq %LAST-YES-NO '?)  
    )  
  )  
;=====
```

```
;===== MENU
```

```

      ;-- (MENU menuname
      ;--      (MSG message)
      ;--      (OPTION symbol
      ;--      {MSG message}
      ;--      (OPTION symbol
      ;--      (MSG message)
      ;--      )
      ;--      ...)

(defun MENU fexpr (args)
  (let (menuname
        YES-choices
        DONT-KNOW-choices
        options
        actions
        (TAB 3))
    (setq menuname (car args))
    (cond
      ((symbol-only args) ;-- (MENU
symbol)
        (if (not (setq actions (get menuname 'MENU)))
            (error "*** undefined menu : " menuname))
        (if (neq (get menuname 'MENU-DONE) VERSION)
            (putprop menuname VERSION 'MENU-DONE)
            )
        (for action in actions
          do
            (cond
              ((eq (car action) 'MSG)
               (msg N) (eval action) (msg N))
              ((eq (car action) 'OPTION)
               (if (not (symbolp (cadr action)))
                   (error "*** missing option-name : " action))
               (nconc-to-list (cadr action) options)
               (msg (C TAB) "(" (length options) ") ")
               (eval (caddr action)) (msg N)
              )
              (else
               (eval action))
            )
          )
        (selectq (input-choices (length options))
          (N
           ;(my-eval 'NEW-ASKED-FACTS `((NOT ,menuname)))
           )
          (Y
           ;(my-eval 'NEW-ASKED-FACTS `((,menuname)))
           (for let (count 1)
             action in actions
             when (eq (car action) 'OPTION)
             do
               (cond
                 ((member count 'YES-choices)
                  (my-eval 'NEW-ASKED-FACTS `((, (cadr
action))))
                  (setq %LAST-YES-NO 'Y)
                  (my-eval 'DO (caddr action))
                  )
                 ((member count 'DONT-KNOW-choices)
                  (my-eval 'NEW-ASKED-FACTS `(('? ,(cadr
action))))
                  (setq %LAST-YES-NO '?)
                  (my-eval 'DO (caddr action))
                  )
               )
             )
           )
          )
        )
      )
    )
  )

```

```

                                (setq count (1+ count))
                            )
                    )
            )
        ((actions-only args)                                     ;-- (MENU
actions)                (error "*** missing menu-name ***")
                        )
        ((symbol-and-actions args)                               ;-- (MENU symbol
actions)                (putprop menuname (cdr args) 'MENU)
                        (push menuname %MENU-list)
                        )
    )
)
t
)

;----- input-choices

;-- inputs a list of, eg. 2 ?4 3 ?1
;-- sets YES-NO-choices to '(2 3)
;-- sets DONT-KNOW-choices to '(4 1)

(defun input-choices (num-options)
  (for let (invalid t)
    (retval 'Y)
    while invalid
    do
      (setq invalid nil)
      (setq YES-choices nil)
      (setq DONT-KNOW-choices nil)
      (let ((choices (readline)))
        (if (and (not (numberp (car choices)))
                  (member (getchar (car choices) 1) '(n N)))
            (setq retval 'N)
            else
              (for let (lname)
                choice in choices
                until invalid
                do
                  (cond
                     ((numberp choice)
                      (setq lname 'YES-choices))
                     ((eq (getchar choice 1) '?)
                      (setq choice (- (cadr (exploden choice)) 48))
                      (setq lname 'DONT-KNOW-choices))
                     )
                  )
                (if (or (not (numberp choice))
                       (< choice 0)
                       (> choice num-options))
                    (setq invalid t)
                    (msg "Invalid choice. Try again." N)
                    else
                      (set lname (cons choice (eval lname))))
                )
              )
          )
        finally
          retval
        )
  )
)
```

```

;===== MSG

(defun MSG fexpr (args)
  (my-eval 'msg args)
  t
)

;===== NEW-ASKED-FACTS

(defun NEW-ASKED-FACTS fexpr (facts)
  (for fact in facts
    do
      (if (neq (get (fname fact) 'ASKED) FACT-VERSION)
        (if (neq (car fact) '?)
          (push fact %ASKED-list))
        )
      (if (eq (car fact) '?)
        (nconc-to-list (cadr fact) %DONT-KNOW-list)
        else
        (nconc-to-list (fname fact) %SUMMARY-list)
        )
      (new-fact fact 'ASKED)
    )
  )

;===== NEW-FACTS

(defun NEW-FACTS fexpr (facts)
  (for fact in facts
    do
      (new-fact fact 'NOT-ASKED)
    )
  t
)

;===== new-fact

(defun new-fact (fact asked-flag)
  (let ((f (fname fact)))
    (if (neq (get f 'VERSION) FACT-VERSION)
      (set f nil)
      (putprop f FACT-VERSION 'VERSION)
    )
    (cond
      ((eq (car fact) 'NOT) (new-NOT-fact f))
      ((eq (car fact) '?) (new-VALUE-fact f '?))
      ((cdr fact) (new-VALUE-fact f (cadr fact)))
      (else (new-TRUE-fact f))
    )

    (mapc 'test-FF-rule (get f 'FF-references))
    (mapc 'test-FA-rule (get f 'FA-references))
  )
)

;=====

(defun new-TRUE-fact (f)
  (if (and (eq (eval f) 'FALSE)
    (eq (get f 'ASKED) FACT-VERSION)
    (neq asked-flag 'ASKED))
    (nconc-to-list f %CONFLICT-list)
    )
  (if (eq asked-flag 'ASKED)
    (putprop f FACT-VERSION 'ASKED)
    )
  (if (neq (eval f) 'TRUE) ; (-- dont do IMPLIES etc if

```

```

already done --)
  (set f 'TRUE)
  (let ((IMPLIES (get f 'IMPLIES)))
    (if IMPLIES
      (if trace
        (msg N "    -- " fact " --> " IMPLIES N))
      (my-eval 'NEW-FACTS IMPLIES))
    )
  )
)

;-----

(defun new-NOT-fact (f)
  (if (and (eq (eval f) 'TRUE)
    (eq (get f 'ASKED) FACT-VERSION)
    (neq asked-flag 'ASKED))
    (nconc-to-list f %CONFLICT-list)
  )
  (if (eq asked-flag 'ASKED)
    (putprop f FACT-VERSION 'ASKED)
  )
  (if (neq (eval f) 'FALSE)      ; {-- dont do IMPLIES etc if
already done --}
    (set f 'FALSE)
    (let ((NOT-IMPLIES (get f 'NOT-IMPLIES)))
      (if NOT-IMPLIES
        (if trace
          (msg N "    -- " fact " --> " NOT-IMPLIES N))
        (my-eval 'NEW-FACTS NOT-IMPLIES))
      )
    )
  )
)

;-----

(defun new-DONT-KNOW-fact (f)
  (if (or (eq (eval f) 'TRUE)
    (eq (eval f) 'FALSE))
    (msg "*** inconsistency : " fact " (previously " (eval f) ") "
N)
  )
  (if (eq asked-flag 'ASKED)
    (putprop f FACT-VERSION 'ASKED)
  )
  (set f '?)
)

;-----

(defun new-VALUE-fact (f value)
  (if (or (eq (eval f) 'TRUE)
    (eq (eval f) 'FALSE))
    (msg "*** inconsistency : " fact " (previously " (eval f) ") "
N)
  )
  (set f value)
)

;===== test-FF-rule

(defun test-FF-rule (FF-name)
  (if (neq (get FF-name 'DONE) FF-VERSION)
    (if (my-eval 'AND-FACTS (get FF-name 'LHS))
      (if trace

```

```

      (msg N "    -- firing " FF-name N
        "      " (get FF-name 'LHS) N
        "      -->" (get FF-name 'RHS) N)
    )
  (putprop FF-name FF-VERSION 'DONE)
  (my-eval 'NEW-FACTS (get FF-name 'RHS))
)
)

;===== test-FA-rule

(defun test-FA-rule (FA-name)
  (if (and (neq (get FA-name 'DONE) VERSION)
           (neq (get FA-name 'READY) VERSION))
      (if (my-eval 'AND-FACTS (get FA-name 'LHS))
          (let ((priority (get FA-name 'PRIORITY)))
            (store (%FA-firelist priority)
                   (insert FA-name (%FA-firelist priority) nil
                           'nodups)))
          (putprop FA-name VERSION 'READY)
          (if trace
              (msg N "    -- adding " FA-name " to priority-" priority "
list." N)))
      )
  )
)

(defun QVALUE (value)
  (NEW-FACTS `(.QUESTION-name ,value))
)

;=====

```

```

;===== QUESTION

(setq QUESTION 'Q)
(defun Q fexpr (args)
  (do-question nil args))
(defun Q* fexpr (args)
  (do-question t args))

(defun do-question (always-do args)
  (let (symbol
        QUESTION-name
        actions)
    (setq symbol (car args))
    (cond
      ((symbol-only args)
        (if (or always-do
                (and (neq (get symbol 'QUESTION-DONE) VERSION)
                     (null (fact-value symbol))))
            ;-- QUESTION-name is on a "stack" (ie. local)
            so
            ;-- that all enclosed YES-NOs can see it.
            (setq QUESTION-name symbol)
            (setq actions (cond ((get symbol 'QUESTION))
                                ((list (list 'YES-NO symbol)))))
            (putprop symbol VERSION 'QUESTION-DONE)
            (my-eval 'DO actions)
            )
          (test-fact `(.symbol)) ; return t if symbol true
        )
      ((actions-only args)
        (msg "*** error : missing question-name : " args N)
        (lispbreak "*** ")
        )
      ((symbol-and-actions args)
        (clear-fact symbol)
        (putprop symbol (cdr args) 'QUESTION)
        (push symbol %QUESTION-list)
        )
    )
  )
)

;----- symbol-only etc.

(defmacro symbol-only (args)
  `(and (symbolp (car ,args))
        (not (cdr ,args)))
  )

(defmacro actions-only (args)
  `(and (not (symbolp (car ,args))))
  )

(defmacro symbol-and-actions (args)
  `(and (symbolp (car ,args))
        (cdr ,args))
  )

;===== SOLUTION

(defun SOLUTION fexpr (args)
  (prog (symbol)
    (setq symbol (car args))
    (cond
      ((symbol-only args) ;-- (SOLUTION

```



```

symbol)
    (msg N "-----" N
      "SOLUTION :" N)
    (eval (cons 'DO (get symbol 'SOLUTION)))
    (msg N "-----" N N)
    (lispbreak "(SOLVED) ")
  )
  ((actions-only args) ;-- (SOLUTION
actions)
    (my-eval 'DO args)
    (lispbreak "(SOLVED) ")
  )
  ((symbol-and-actions args) ;-- (SOLUTION symbol
actions)
    (clear-fact symbol)
    (putprop symbol (cdr args) 'SOLUTION)
    (push symbol %SOLUTION-list)
  )
)
)
t
)

```

===== SUMMARY

```

(defun SUMMARY ()
  (prog (desc)
    (msg N "-----" N
      "The following is a summary of what you have told me so
far :" N)
    (for bind (count 1 (add1 count))
      (pcount 1)
      symbol in %SUMMARY-list
      do
        (if (setq desc (get symbol 'DESC))
          (msg " (" pcount ") ")
          (print-DESC symbol) (msg N)
          (setq pcount (+ 1 pcount)))
        ;else
        ; (print-DESC symbol) (msg N) ;*** remove later
        )
      )
    (msg N "Which of these are incorrect ?" N)
    (msg " (enter 0 if none)" N)
    (input-choices (length %SUMMARY-list))
    (if (nequal YES-choices '(0))
      (for choice in YES-choices
        do
          (msg "--- knowledge-base-rebuild not working yet ---" N)
          ;*** put in from 'correct-conflicts', when it is working
          )
      )
    )
    (msg "-----" N)
  )
  t
)

```

===== YES-NO

```

(defun YES-NO fexpr (args)
  (setq YN 'YES-NO)
  (my-eval 'YESNO args)
)

```

```

(defun NO-YES fexpr (args)
  (setq YN 'NO-YES)
)

```

```

(my-eval 'YESNO args)
)

;===== YESNO

(defun YESNO fexpr (args)
  (prog (answer
        symbol
        message)
    (cond
      ((symbolp (car args)) ;-- (YES-NO
symbol message)
      (setq symbol (car args)) ;-- (if message
is null
      (setq message (cdr args))) ;-- then use
"symbol")
      (else ;-- (YES-NO
message)
      (setq symbol QUESTION-name)
      (setq message args))
    )
    (putprop symbol message 'YES-NO-msg)

    ask-question
    (if message
      (msg N) (eval (append '(MYMSG rub) message)) (msg " " N)
      else
      (msg N) (msg symbol " ?" N)
    )
    wait-for-response
    (setq %LAST-YES-NO nil)
    (setq answer (read))
    (selectq (getchar answer 1)
      ((y Y) (setq %LAST-YES-NO 'Y)
      (if (eq YN 'YES-NO) (my-eval 'NEW-ASKED-FACTS
'((.symbol))))
      else
      (my-eval 'NEW-ASKED-FACTS `((NOT
.symbol))))
      )
      ((n N) (setq %LAST-YES-NO 'N)
      (if (eq YN 'YES-NO) (my-eval 'NEW-ASKED-FACTS
'((NOT .symbol)))
      else
      (my-eval 'NEW-ASKED-FACTS
'((.symbol))))
      )
      (?
      (setq %LAST-YES-NO '?)
      (my-eval 'NEW-ASKED-FACTS `((? .symbol)))
      )
      ((q Q) (msg N N "*** diagnosis aborted ***" N N)
      (lispbreak "*** "))
      (otherwise
      (let ((op (substring answer 1 3)))
        (cond
          ((member op '("SUM" "sum")) (SUMMARY))
          ((member op '("WHY" "why" "SOL" "sol"))
(HYPOTHESIS))
          (otherwise (msg " *** invalid answer. try again
***" N)
          (go wait-for-response))
        )
      )
      (go ask-question)
    )
    (return %LAST-YES-NO)
  )

```

```

)

;===== YES-FACTS etc

(defun YES-FACTS fexpr (facts)
  (if (eq %LAST-YES-NO 'Y)
      (my-eval 'NEW-FACTS facts)
      )
  )

(defun NO-FACTS fexpr (facts)
  (if (eq %LAST-YES-NO 'N)
      (my-eval 'NEW-FACTS facts)
      )
  )

(defun YES-NO-FACTS fexpr (facts)
  (if (eq %LAST-YES-NO 'Y)
      (my-eval 'NEW-FACTS facts)
      )
  (if (eq %LAST-YES-NO 'N)
      (my-eval 'NEW-FACTS (not-facts facts))
      )
  )

;=====

```

```

;===== fname

;--      returns the fact-name of fact
;--      'fact' is in form :   '(fact)
;--                               or '(NOT fact)

(defmacro fname (fact)
  `(cond ((cadr ,fact))
        ((car ,fact)))
  )

;===== fact?

;--      returns t if action is '(factname)
;--                               '(NOT factname)
;--                               ' (? factname)

(defmacro fact? (action)
  `(or (not-fact? ,action)
      (true-fact? ,action)
      (dont-know-fact? ,action)
      )
  )

;===== true-fact?

(defun true-fact? (action)
  (and (symbolp (car action))
       (not (cdr action))
       (not (member (car action)
                     '(SUMMARY HYPOTHESES IF-NOVICE IF-AVERAGE
IF-EXPERT))))
  )
  )

;===== not-fact?

(defmacro not-fact? (action)
  `(and (eq (car ,action) 'NOT)
       (symbolp (cadr ,action)))
  )

;===== dont-know-fact?

(defmacro dont-know-fact? (action)
  `(and (eq (car ,action) '?)
       (symbolp (cadr ,action)))
  )

;===== test-fact-expr

(defun test-fact-expr (fact-expr)
  (cond
    ((or (not (cdr fact-expr)) ;-- '(factname)
         (symbolp (cadr fact-expr))) ;-- '(NOT factname) ' (?
factname)
      (test-fact fact-expr)
      )
    ((eq (car fact-expr) 'AND)
      (my-eval 'AND-FACTS (cdr fact-expr))
      )
    ((eq (car fact-expr) 'OR)
      (my-eval 'OR-FACTS (cdr fact-expr))
      )
    ((eq (car fact-expr) 'NOT)
      (my-eval 'NOT-FACTS (cdr fact-expr))
      )
  )
  )

```

```

    )
  (t
    (msg "*** illegal fact-clause : " fact-expr N)
    (lispbreak "*** ")
  )
)

;===== test-fact

(defun test-fact (fact)
  (let ((f (fname fact)))
    (and (eq (get f 'VERSION) FACT-VERSION)
      (cond
        ((not (cdr fact)) (eq (eval f) 'TRUE))
        ((eq (car fact) 'NOT) (or (eq (eval f) 'FALSE)
                                   (eq (eval f) '?)))
        ((eq (car fact) '?) (eq (eval f) '?))
        (else (equal (eval f) (cadr fact))))
      )
    )
  )

;===== fact-value

(defmacro fact-value (factname)
  `(cond ((neq (get ,factname 'VERSION) FACT-VERSION)
        nil)
        ((eval ,factname))
  )
)

;===== get-fact-names

(defun get-fact-names (fact-exprs)
  (cond
    ((null fact-exprs)
     nil
    )
    ((listp (car fact-exprs)) ;-- '((...) (...))
     (append (get-fact-names (car fact-exprs))
              (get-fact-names (cdr fact-exprs)))
    )
    ((not (cdr fact-exprs)) ;-- '(factname)
     fact-exprs
    )
    ((symbolp (cadr fact-exprs)) ;-- '(NOT factname) '(?
factname)
     (cdr fact-exprs)
    )
    ((member (car fact-exprs) '(AND OR NOT))
     (get-fact-names (cdr fact-exprs))
    )
  )
  (t
    (msg "*** invalid facts : " fact-exprs N)
    (lispbreak "*** ")
  )
)

;===== not-facts

(defun not-facts (facts)
  ; {-- facts is in form '((fact) (NOT fact)
  ..) --}

```

```

                                ; {-- returns a list of the facts NOTed --}
  (if (not (null facts))
      (cons (not-fact (car facts))
            (not-facts (cdr facts)))
      )
  )

;-----

(defun not-fact (fact)
  (cond
    ((eq (car fact) 'NOT) (cdr fact))
    ((eq (car fact) '?) fact)
    (t (cons 'NOT fact))
  )
)

;===== addprop

;-----
;      'values' is a list of items which are to be added
;      to the property list.
;-----

(defun addprop (name values prop)
  (putprop name (append values (get name prop)) prop)
)

;===== nconc-to-list

(defun nconc-to-list (item lname)
  `(setq ,lname (append ,lname (list ,item)))
)

;===== push

(defun push (item lname)
  `(setq ,lname (cons ,item ,lname))
)

;===== error

(defun error fexpr (message)
  (msg N) (my-eval 'msg message) (msg N N)
  (lispbreak "*** ")
)

;=====

```

```

;===== LOAD-FILES

(defun LOAD-FILES fexpr (files)
  (setq LOAD-VERSION (newsym 'V))
  (initsym '(FA-rule 1000)) ;-- use 1000 so insert gets right
  order
  (initsym '(FF-rule 0))
  (setq %FACT-list nil
        %FA-list nil
        %FF-list nil
        %MENU-list nil
        %QUESTION-list nil
        %SOLUTION-list nil
        )
  (mapc 'load-FILE files)
  (msg N "---- loaded ----" N N)
  t
)

```

```

;-----

(defun load-FILE (file)
  (msg N " -- loading file : " file N)
  (let ((port (fileopen file 'r)))
    (for being (action (read port))
      while action
      do
        (msg ".")
        (eval action)
      )
    (close port)
  )
)

```

```

;===== clear-facts

```

```

(defun clear-facts (fact-exprs)
  (for f in (get-fact-names fact-exprs)
    do
      (clear-fact f)
  )
)

(defun clear-fact (f)
  (if (neq (get f 'VERSION) LOAD-VERSION)
    (setplist f nil)
    (putprop f LOAD-VERSION 'VERSION)
    (setq %FACT-list (nconc %FACT-list `(.f)))
  )
)

```

```

;===== FA

```

```

(defun FA fexpr (rule)
  (prog (FA-name priority LHS ACTIONS)
    (setq FA-name (newsym 'FA-rule))
    (setplist FA-name nil)
    (if (numberp (car rule))
      (setq priority (car rule)
              LHS (left-side (cdr rule))
              RHS (right-side (cdr rule)))
      else
      (setq priority 0
              LHS (left-side rule)
              RHS (right-side rule))
    )
  )
)

```

```

    (clear-facts LHS)
    (putprop FA-name priority 'PRIORITY)
    ;(putprop FA-name LOAD-VERSION 'VERSION)
    (putprop FA-name LHS 'LHS)
    (putprop FA-name RHS 'RHS)
    (setq %FA-list (nconc %FA-list (list FA-name)))
    (add-fact-references LHS FA-name 'FA-references)
  )
)

;===== FF

(defun FF fexpr (rule)
  (define-FF rule 'reverse)
)

(defun FF* fexpr (rule)
  (define-FF rule 'dont-reverse)
)

(defun define-FF (rule FF-type)
  (prog (LHS RHS)
    (setq LHS (left-side rule)
          RHS (right-side rule))
    (clear-facts LHS)
    (clear-facts RHS)

    ;-----

    ;-- if one '(fact) or '(NOT fact) in LHS
    ;-- then put in IMPLIES list
    (let ((fact (car LHS))
          (not-left))
      (if (and (eq (length LHS) 1)
              (or (not (cdr fact))
                  (eq (cadr fact) 'NOT)))
          )
      (cond
        ((null (cdr fact)) (addprop (car fact) RHS 'IMPLIES)
                          (setq not-left (cons 'NOT fact)))
        ((eq (car fact) 'NOT) (addprop (cadr fact) RHS
                                      'NOT-IMPLIES)
                          (setq not-left (cadr fact)))
      )

      ;-- now add the reverse
      (if (eq FF-type 'reverse)
          (for RHS-fact in RHS
            do
              (cond
                ((eq (car RHS-fact) 'NOT)
                 (addprop (cadr RHS-fact) `(.not-left)
                          'IMPLIES))
                ((not (cdr RHS-fact))
                 (addprop (car RHS-fact) `(.not-left)
                          'NOT-IMPLIES))
              )
            )
          )
      )

    ;-----

    else ;-- several facts in LHS, so store in a
    rule ;-- and put its name in FF-rules list.
    (let ((FF-name (newsym 'FF-rule)))

```



```

        (putprop FF-name LOAD-VERSION 'VERSION)
        (setq %FF-list (nconc %FF-list (list FF-name)))
        (putprop FF-name LHS 'LHS)
        (putprop FF-name RHS 'RHS)
        (add-fact-references LHS FF-name 'FF-references)
      )
    )
  )
)

;===== add-fact-references

(defun add-fact-references (fact-exprs refname reifprop)
  (for factname in (get-fact-names fact-exprs)
    do
      (addprop factname `(.refname) reifprop)
    )
  )

;=====
left-side

(defun left-side (rule)
  (if (null rule)
    (msg "*** missing '-->' ***" N N)
    (lispbreak "*** ")
  )
  (if (eq (car rule) '-->)
    nil
    else
      (cons (car rule) (left-side (cdr rule)))
    )
  )

;===== right-side

(defun right-side (rule)
  (cdr (member '--> rule))
)

;===== print-rules

(defun print-rules ()
  (print-FACTS)
  (print-FF-rules)
  (print-FA-rules)
  (print-ACTIONS)
)

(defun print-FACTS ()
  (msg N N "----- FACTS -----" N)
  (for f in (sort %FACT-list nil)
    do
      (msg N f)
      (if (eq (get f 'VERSION) FACT-VERSION)
        (msg " = " (eval f)))
      (msg N)
      (print-props f 3 '(DESC ASKED IMPLIES NOT-IMPLIES
        FF-references FA-references
        MENU MENU-DONE
        QUESTION QUESTION-DONE
        SOLUTION SOLUTION-DONE))
    )
  )
)

```

```

(defun print-FF-rules ()
  (msg N N "----- FF-rules -----" N)
  (for rule in %FF-list
    do
      (msg N rule N)
      (print-props rule 3 '(LHS RHS DONE))
    )
  )

(defun print-FA-rules ()
  (msg N N "----- FA-rules -----" N)
  (for rule in %FA-list
    do
      (msg N rule N)
      (print-props rule 3 '(LHS RHS DONE))
    )
  )

(defun print-ACTIONS ()
  (for atype in '(MENU QUESTION SOLUTION)
    do
      (msg N N "----- " atype " -----" N)
      (for aname in (sort (eval (concat '% atype '-list)) nil)
        do
          (msg N aname N)
          (print-actions atype aname 3)
        )
      )
    )
  )

```

```

;-----

(defun print-props (name col props)
  (for prop in props
    when (get name prop)
    do
      (msg (C col) prop " : ")
      (if (<= (length (explode prop)) 13)
        (msg (C 19)))
      (msg (get name prop) N)
    )
  )

```

```

;-----

(defun print-actions (atype aname col)
  (msg (C col) )
  (for action in (get aname atype)
    do
      (msg (C col) action N)
    )
  )

```

```

;=====

```